Continuation-Passing C, compiling threads to events through continuations

Gabriel Kerneis, Juliusz Chroboczek

▶ **To cite this version:**

Gabriel Kerneis, Juliusz Chroboczek. Continuation-Passing C, compiling threads to events through continuations. 2012. ⟨inria-00537964v2⟩

HAL Id: inria-00537964

https://inria.hal.science/inria-00537964v2

Preprint submitted on 14 Feb 2012 (v2), last revised 5 Jul 2012 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Higher-Order Symb Comput manuscript No.
(will be inserted by the editor)

Continuation-Passing C

Compiling threads to events through continuations

Gabriel Kerneis · **Juliusz Chroboczek**

Received: date / Accepted: date

Abstract In this paper, we introduce Continuation Passing C (CPC), a programming language for concurrent systems in which native and cooperative threads are unified and presented to the programmer as a single abstraction. The CPC compiler uses a compilation technique, based on the CPS transform, that yields efficient code and an extremely lightweight representation for contexts. We provide a proof of the correctness of our compilation scheme. We show in particular that lambda-lifting, a common compilation technique for functional languages, is also correct in an imperative language like C, under some conditions enforced by the CPC compiler. The current CPC compiler is mature enough to write substantial programs such as Hekate, a highly concurrent BitTorrent seeder. Our benchmark results show that CPC is as efficient, while using significantly less space, as the most efficient thread libraries available.

Keywords Concurrent programming · Lambda-lifting · Continuation-Passing Style

Contents

1 Introduction 1
2 Related work 3
3 The CPC language 5
4 Experimental results 9
5 The CPC compilation technique 13
6 Lambda-lifting in an imperative language 25
7 Conclusions and further work 38

1 Introduction

Most computer programs are *concurrent* programs, which need to perform multiple tasks at a given time. For example, a network server needs to serve multiple clients at a time;

G. Kerneis
Laboratoire PPS, Université Paris Diderot, Case 7014, 75205 Paris Cedex 13, France
E-mail: kerneis@pps.jussieu.fr

J. Chroboczek
Laboratoire PPS, Université Paris Diderot, Case 7014, 75205 Paris Cedex 13, France

a program with a graphical user interface needs to expect keystrokes and mouse clicks at multiple places; and a network program with a graphical interface (e.g. a web browser) needs to do both.

The dominant abstraction for concurrency is provided by *threads*, or *lightweight processes*. When programming with threads, a process is structured as a dynamically changing number of independently executing threads that share a single heap (and possibly other data, such as global variables). The alternative to threads is *event-loop* or *event-driven* programming. An event-driven program interacts with its environment by reacting to a set of stimuli called *events*. At any given point in time, to every event is associated a piece of code known as the *handler* for this event. A global scheduler, known as the *event loop*, repeatedly waits for an event to occur and invokes the associated handler.

Unlike threads, event handlers do not have an associated stack; event-driven programs are therefore more lightweight and often faster than their threaded counterparts. However, because it splits the flow of control into multiple tiny event-handlers, event-driven programming is difficult and error-prone. Additionally, event-driven programming alone is often not powerful enough, in particular when accessing blocking APIs or using multiple processor cores; it is then necessary to write *hybrid* code, that uses both threads and event handlers, which is even more difficult.

Since event-driven programming is more difficult but more efficient than threaded programming, it is natural to want to at least partially automate it.

Continuation Passing C Continuation Passing C (CPC) is an extension of the C programming language with concurrency primitives which is implemented by a series of source-to-source transformations, including a phase of lambda-lifting (Section 5.5) and a transformation into Continuation-Passing Style (Section 5.2). After this series of transformations, the CPC translator yields a program in hybrid style, most of it event-driven, but using the native threads of the underlying operating system wherever this has been specified by the CPC programmer, for example in order to call blocking APIs or distribute CPU-bound code over multiple cores or processors.

Although it is implemented as a series of well-understood source-to-source translations, CPC yields code that is just as fast as the fastest thread libraries available to us; as a matter of fact, we believe that the code it generates is similar to carefully tuned code written by a human programmer with a lot of experience with event-driven code. And although it is an extension of the C programming language, a language notoriously hostile to formal proofs, the correctness of the transformations used by CPC have been proved; while we perform these proofs in a simplified dialect, there is nothing in principle that prevents a complete proof of the correctness of CPC. We believe that this property makes CPC unique among the systems for efficient implementation of concurrency in a widely used imperative programming language.

In this paper, we give an introduction to CPC programming, we describe informally the transformations performed by CPC, and give an outline of the proof of correctness; the complete proof can be found in a companion technical report [28].

1.1 Outline

The rest of this paper is organised as follows. In Section 2, we present previous work related to threads and event-driven programming, and to the compilation techniques used by the CPC translator. In Section 3, we give an overview of the CPC language through excerpts

of code from Hekate, the most significant program written in CPC so far. In Section 4, we show that CPC threads are as fast as the most efficient thread libraries available to us. In Section 5, we detail the several passes of the CPC compilation technique to translate threads into events, and in Section 6 we prove the correctness of one of these passes, lambda-lifting, in the context of an imperative, call-by-value language restricted to functions called in tail position. We conclude in Section 7.

2 Related work

2.1 Continuations and concurrency

Continuations offer a natural framework to implement concurrency systems in functional languages: they capture "the rest of the computation" much like the state of an imperative program is captured by the call stack. Thread-like primitives may be built with either first-class continuations, or encapsulated within a continuation monad.

The former approach is best illustrated by *Concurrent ML* constructs [38], implemented on top of SML/NJ's first-class continuations, or by the way coroutines and engines are typically implemented in Scheme using the `call/cc` operator [22, 15] (previously known as `catch` [52]). *Stackless Python* [49] uses first-class continuations to implement generators, which are in turn used to implement concurrency primitives. Scala also uses first-class continuations, through the `shift` and `reset` operators, to implement concurrency primitives and asynchronous IO [40].

Explicit translation into continuation-passing style, often encapsulated within a monad, is used in languages lacking first-class continuations. In Haskell, the original idea of a concurrency monad is due to Scholz [41], and extended by Claessen [9] to a monad transformer yielding a concurrent version of existing monads. Li et. al [33] use a continuation monad to lazily translate the thread abstraction exposed to the programmer into events scheduled by an event loop. In Objective Caml, Vouillon's *lwt* [50] provides a lightweight alternative to native threads, with the ability to execute blocking tasks in a separate thread pool.

2.2 Transformation techniques

The three main techniques used by CPC — CPS conversion, lambda-lifting and splitting — are fairly standard techniques for compiling functional languages, or at least languages providing inner functions.

The conversion into continuation-passing style, or CPS conversion, has been discovered and studied many times in different contexts [37, 46, 39]. It is used for instance to compile Scheme (Rabbit [44]) and ML (SML/NJ [2]), both of them exposing continuations to the programmer.

Lambda-lifting, also called closure-conversion, is a standard technique to remove free variables. It is introduced by Johnsson [25] and made more efficient by Danvy and Schultz [11]. Fischbach and Hannan prove its correctness for the Scheme language[17]. Although environments are a more common way to handle free variables, some implementations use lambda-lifting; for instance, the *Twobit* Scheme-to-C compiler [10].

We call *splitting* the conversion of a complex flow of control into mutually recursive function calls. Van Wijngaarden is the first to describe such a transformation, in a source-to-source translation for Algol 60 [54]. The idea is then used by Landin to formalise a

translation between Algol and the lambda-calculus [31], and by Steele and Sussman to express gotos in applicative languages such as LISP or Scheme [44]. Thielecke adapts van Wijngaarden's transformation to the C language, albeit in a restrictive way [48].

We are aware of two implementations of splitting: Weave and TameJS. *Weave* is an unpublished tool used at IBM around year 2000 to write firmware and drivers for SSA-SCSI RAID storage adapters [29]. It translates annotated Woven-C code, written in threaded style, into C code hooked into the underlying event-driven kernel. *TameJS*[1], by the authors of the C++ *Tame* library [30], is a similar tool for Javascript where event-driven programming is made mandatory by the lack of concurrency primitives.[2]

2.3 Threads and events

There has been a lot of research to provide efficient threads, as well as to make event-driven programming easier [4, 53, 36]. We focus here on results involving a transformation between threads and events, or building bridges between them.

Adya et al. [1] present adaptors between event-driven and threaded code to write programs mixing both styles. They first introduce the notion of *stack ripping* to describe the manual stack management required by event-driven style.

Duff introduces a technique, known as *Duff's device* [13], to express general loop unrolling directly in C, using the `switch` statement. Since then, this technique has been employed multiple times to express state machines and event-driven programs in a threaded style.[3] For instance, it is used by Tatham to implement coroutines in C [47]. Other C libraries later expanded this idea, such as *protothreads* [14] and *FairThreads*' automata [6]. These libraries help keep a clearer flow of control but they provide no automatic handling of local variables: the programmer is expected to save them manually in his own data structures, just like in event-driven style.

Tame [30] is a C++ language extension and library which exposes events to the programmer while introducing state machines to avoid the stack ripping issue and retain a thread-like feeling. The programmer needs to annotate local variables that must be saved across context switches.

TaskJava [18] implements the same idea than Tame, in Java, but preserves local variables automatically, storing them in a state record. *Kilim* [43] is a message-passing framework for Java providing actor-based, lightweight threads. It is also implemented by a partial CPS conversion performed on annotated functions, but contrary to TaskJava, it works at the JVM bytecode level.

AC is a set of language constructs for composable asynchronous IO in C and C++ [21]. Harris et al. introduce `do..finish` and `async` operators to write asynchronous requests in a synchronous style, and give an operational semantics. The language constructs are somewhat similar to those of Tame but the implementation is widely different, using LLVM code blocks or macros based on GCC nested functions rather than source-source transformations.

Haller and Odersky [20] advocate unification of thread-based and event-based models through actors, with the `react` and `receive` operators provided by the *Scala Actors* library; suspended actors are represented by continuations.

[1] `http://tamejs.org/`

[2] Note that, contrary to TameJS, the original Tame implementation in C++ does not use splitting but a state machine using switches.

[3] This abuse was already envisioned by Duff in 1983: "I have another revolting way to use switches to implement interrupt driven state machines but it's too horrid to go into." [13]

3 The CPC language

Together with two undergraduate students, we taught ourselves how to program in CPC by writing *Hekate*, a *BitTorrent seeder*, a massively concurrent network server designed to efficiently handle tens of thousands of simultaneously connected peers [27, 3]. In this section, we give an overview of the CPC language through a number of programming idioms that we discovered while writing Hekate.

3.1 Cooperative CPC threads

The extremely lightweight, cooperative threads of CPC lead to a "threads are everywhere" feeling that encourages a somewhat unusual programming style.

Lightweight threads Contrary to the common model of using one thread per client, Hekate spawns at least three threads for every connecting peer: a reader, a writer, and a timeout thread. Spawning several CPC threads per client is not an issue, especially when only a few of them are active at any time, because idle CPC threads carry virtually no overhead.

The first thread reads incoming requests and manages the state of the client. The BitTorrent protocol defines two states for interested peers: "*unchoked*," i.e. currently served, and "*choked*." Hekate maintains 90 % of its peers in *choked* state, and *unchokes* them in a round-robin fashion.

The second thread is in charge of actually sending the chunks of data requested by the peer. It usually sleeps on a condition variable, and is woken up by the first thread when needed. Because these threads are scheduled cooperatively, the list of pending chunks is manipulated by the two threads without need for a lock.

Every read on a network interface is guarded by a timeout, and a peer that has not been involved in any activity for a period of time is disconnected. Earlier versions of Hekate which did not include this protection would end up clogged by idle peers, which prevented new peers from connecting.

In order to simplify the protocol-related code, timeouts are implemented in the buffered read function, which spawns a new timeout thread on each invocation. This temporary third thread sleeps for the duration of the timeout, and aborts the I/O if it is still pending. Because most timeouts do not expire, this solution relies on the efficiency of spawning and context-switching short-lived CPC threads (see Section 4).

Native and cps functions CPC threads might execute two kinds of code: *native* functions and *cps* functions (annotated with the `cps` keyword). Intuitively, cps functions are interruptible and native functions are not: it is possible to interrupt the flow of a block of cps code in order to pass control to another piece of code, to wait for an event to happen or to switch to another scheduler (Section 3.3). Native code, on the other hand, is "atomic": if a sequence of native code is executed in attached state, it must be completed before anything else is allowed to run in the same scheduler. From a more technical point of view, cps functions are compiled by performing a transformation to Continuation Passing Style (CPS), while native functions execute on the native stack.

There is a global constraint on the call graph of a CPC program: a cps function may only be called by a cps function; equivalently, a native function can only call native functions — but a cps function may call a native function. This means that at any point in time, the dynamic chain consists of a "cps stack" of cooperating functions followed by a "native

```
cps void
listening(hashtable * table) {
    /* ... */
    while(1) {
        cpc_io_wait(socket_fd, CPC_IO_IN);
        client_fd = accept(socket_fd, ...);
        cpc_spawn client(table, client_fd);
    }
}
```

Fig. 1 Accepting connections and spawning threads

stack" of regular C functions. Since context switches are forbidden in native functions, only the cps stack needs to be saved and restored when a thread cooperates.

CPC primitives CPC provides a set of primitive cps functions, which allow the programmer to schedule threads and wait for some events. These primitive functions could not have been defined in user code: they must have access to the internals of the scheduler to operate. Since they are cps functions, they can only be called by another cps function.

Figure 1 shows an example of a cps function: `listening` calls the primitive `cpc_io_wait` to wait for the file descriptor `socket_fd` to be ready, before accepting incoming connections with the native function `accept` and spawning a new thread for each of them. Note the use of the `cpc_spawn` keyword to create a new thread executing the cps function `client`.

The CPC language provides four cps primitives to suspend and synchronise threads on some events. The simplest one is `cpc_yield`, which yields control to the next thread to be executed. The primitives `cpc_io_wait` and `cpc_sleep` suspend the current thread until a given file descriptor has data available or some time has elapsed, respectively. A thread can wait on some condition variable [23] with `cpc_wait` ; threads suspended on a condition variable are woken with the (non-cps) functions `cpc_signal` and `cpc_signal_all`. To allow early interruptions, `cpc_io_wait` and `cpc_sleep` also accept an optional condition variable which, if signaled, will wake up the waiting thread.

A fifth cps primitive, `cpc_link`, is used to control how threads are scheduled. We give more details about it in Section 3.3.

We found that these five primitives are enough to build more complex synchronisation constructs and cps functions, such as barriers or retriggerable timeouts. Some of these generally useful functions, written in CPC and built above the CPC primitives, are distributed with CPC and form the CPC standard library.

3.2 Comparison with event-driven programming

Code readability Hekate's code is much more readable than its event-driven equivalents. Consider for instance the BitTorrent handshake, a message exchange occurring just after a connection is established. In *Transmission*[4], a popular and efficient BitTorrent client written in (mostly) event-driven style, the handshake is a complex piece of code, spanning over a thousand lines in a dedicated file. By contrast, Hekate's handshake is a single function of less than fifty lines including error handling.

While some of Transmission's complexity is explained by its support for encrypted connexions, Transmission's code is intrinsically much more messy due to the use of callbacks

[4] `http://www.transmissionbt.com`

and a state machine to keep track of the progress of the handshake. This results in an obfuscated flow of control, scattered through a dozen of functions (excluding encryption-related functions), typical of event-driven code.

Expressivity Surprisingly enough, CPC threads turn out to be more expressive than native threads, and allow some idioms that are more typical of event-driven style.

A case in point: buffer allocation for reading data from the network. When a native thread performs a blocking read, it needs to allocate the buffer before the `read` system call; when many threads are blocked waiting for a read, these buffers add up to a significant amount of storage. In an event-driven program, it is possible to delay allocating the buffer until after an event indicating that data is available has been received.

The same technique is not only possible, but actually natural in CPC: buffers in Hekate are only allocated after `cpc_io_wait` has successfully returned. This provides the reduced storage requirements of an event-driven program while retaining the linear flow of control of threads.

3.3 Detached threads

While cooperative, deterministically scheduled threads are less error-prone and easier to reason about than preemptive threads, there are circumstances in which native operating system threads are necessary. In traditional systems, this implies either converting the whole program to use native threads, or manually managing both kinds of threads.

A CPC thread can switch from cooperative to preemptive mode at any time by using the `cpc_link` primitive (inspired by FairThreads' `ft_thread_link` [6]). A cooperative thread is said to be *attached* to the default scheduler, while a preemptive one is *detached*.

The `cpc_link` primitive takes a single argument, a scheduler, either the default event loop (for cooperative scheduling) or a thread pool (for preemptive scheduling). It returns the previous scheduler, which makes it possible to eventually restore the thread to its original state. Syntactic sugar is provided to execute a block of code in attached or detached mode (`cpc_attached`, `cpc_detached`).

Hekate is written in mostly non-blocking cooperative style; hence, Hekate's threads remain attached most of the time. There are a few situations, however, where the ability to detach a thread is needed.

Blocking OS interfaces Some operating system interfaces, like the `getaddrinfo` DNS resolver interface, may block for a long time (up to several seconds). Although there exist several libraries which implement equivalent functionality in a non-blocking manner, in CPC we simply enclose the call to the blocking interface in a `cpc_detached` block (see Figure 2a).

Figure 2b shows how `cpc_detached` is expanded by the translator into two calls to `cpc_link`. Note that CPC takes care to attach the thread before returning to the caller function, even though the `return` statement is inside the `cpc_detached` block.

Blocking library interfaces Hekate uses the *curl* library [5] to contact BitTorrent *trackers* over HTTP. Curl offers both a simple, blocking interface and a complex, asynchronous one. We decided to use the one interface that we actually understand, and therefore call the blocking interface from a detached thread.

[5] `http://curl.haxx.se/libcurl/`

```
cpc_detached {
    rc = getaddrinfo(name, ...)
    return rc;
}
```

(a)

```
cpc_scheduler *s =
    cpc_link(cpc_default_threadpool);
rc = getaddrinfo(name, ...)
cpc_link(s);
return rc;
```

(b)

Fig. 2 Expansion of `cpc_detached` in terms of `cpc_link`

```
        prefetch(source, length);                          /* (1) */
        cpc_yield();                                       /* (2) */
        if(!incore(source, length)) {                      /* (3) */
            cpc_yield();                                   /* (4) */
            if(!incore(source, length)) {                  /* (5) */
                cpc_detached {                             /* (6) */
                    rc = cpc_write(fd, source, length);
                }
                goto done;
            }
        }
        rc = cpc_write(fd, source, length);                /* (7) */
    done:
        ...
```

The functions `prefetch` *and* `incore` *are thin wrappers around the* `posix_madvise` *and* `mincore` *system calls.*

Fig. 3 An example of hybrid programming (non-blocking read)

Parallelism Detached threads make it possible to run on multiple processors or processor cores. Hekate does not use this feature, but a CPU-bound program would detach computationally intensive tasks and let the kernel schedule them on several processing units.

3.4 Hybrid programming

Most realistic event-driven programs are actually *hybrid* programs [36, 53]: they consist of a large event loop, and a number of threads (this is the case, by the way, of the *Transmission* BitTorrent client mentioned above). Such blending of native threads with event-driven code is made very easy by CPC, where switching from one style to the other is a simple matter of using the `cpc_link` primitive.

This ability is used in Hekate for dealing with disk reads. Reading from disk might block if the data is not in cache; however, if the data is already in cache, it would be wasteful to pay the cost of a detached thread. This is a significant concern for a BitTorrent seeder because the protocol allows requesting chunks in random order, making kernel readahead heuristics inefficient.

The actual code is shown in Figure 3: it sends a chunk of data from a memory-mapped disk file over a network socket. In this code, we first trigger an asynchronous read of the on-disk data (1), and immediately yield to threads servicing other clients (2) in order to give the kernel a chance to perform the read. When we are scheduled again, we check whether the read has completed (3); if it has, we perform a non-blocking write (7); if it hasn't, we yield one more time (4) and, if that fails again (5), delegate the work to a native thread which can block (6).

Note that this code contains a race condition: the prefetched block of data could have been swapped out before the call to `cpc_write`, which would stall Hekate until the write

completes. Moreover, this race condition is even more likely to appear as load increases and on devices with constrained resources; unfortunately, it cannot be avoided since the Linux kernel does not provide asynchronous disk reads.

Note further that the call to `cpc_write` in the `cpc_detached` block (6) could be replaced by a call to `write`: we are in a native thread here, so the non-blocking wrapper is not needed. However, the CPC primitives such as `cpc_io_wait` are designed to act sensibly in both attached and detached mode; this translates to more complex functions built upon them, and `cpc_write` simply behaves as `write` when invoked in detached mode. For simplicity, we choose to use the CPC wrappers throughout our code.

4 Experimental results

The CPC language provides no more than half a dozen very low-level primitives. The standard library and CPC programs are implemented in terms of this small set of operations, and are therefore directly dependent on their performance. In Section 4.1, we show the results of benchmarking individual CPC primitives against other thread libraries. In these benchmarks, CPC turns out to be comparable to the fastest thread libraries available to us.

In the absence of reliable information on the relative dynamic frequency of cps function calls and thread synchronisation primitives in CPC programs, it is not clear what the performance of individual primitives implies about the performance of a complete program. In Section 4.2, we present the results of benchmarking a set of naive web servers written in CPC and in a number of thread libraries.

Finally, in Section 4.3, we present a number of insights that we gained by working with *Hekate*, our BitTorrent seeder written in CPC.

We compare CPC with the following thread libraries: *nptl*, the native thread library in GNU libc 2.13 (or μClibc 0.9.32 for our tests on embedded hardware) [12]; *GNU Pth* version 2.0.7 [16]; *State Threads (ST)* version 1.9 [42]. *Nptl* is a kernel thread library, while *GNU Pth* and *ST* are cooperative user-space thread libraries.

4.1 Speed of CPC primitives

4.1.1 Space utilisation

On a 64-bit machine with 4 GB of physical memory and no swap space, CPC can handle up to 50.1 million simultaneous threads, which implies an average memory usage of roughly 82 bytes per continuation. This figure compares very favourably to both kernel and user-space thread libraries (see Fig. 4), which our tests have shown to be limited on the same system to anywhere from 32 000 to 934 600 threads in their default configuration, and to 961 400 threads at most after some tuning.

4.1.2 Speed

Fig. 5 presents timings on three different processors: the *Core 2 Duo*, a superscalar out-of-order 64-bit processor, the *Pentium-M*, a slightly less advanced out-of-order 32-bit processor, and the *MIPS 4Kc*, a simple in-order 32-bit RISC chip with a short pipeline.

Library	Number of threads
nptl	32 330
Pth	700 000 (est.) [a]
ST	934 600
ST (4 kB stacks)	961 400
CPC	50 190 000

Fig. 4 Number of threads possible with 4 GB of memory

Library	loop	call[b]	cps-call	switch	cond	spawn
nptl (1 core)	2	2		439	1 318	4 000 000 [c]
nptl (2 cores)	2	2		135	1 791	11 037
Pth	2	2		2 130	2 602	6 940
ST	2	2		170	25	411
CPC	2	2	20	16	36	74

x86-64: Intel Core 2 Duo at 3.17 GHz, Linux 2.6.39

Library	loop	call	cps-call	switch	cond	spawn
nptl	2	4		691	2 426	24 575
ST	2	4		1 003	96	2 293
CPC	2	4	54	46	91	205

x86-32: Intel Pentium M at 1.7 GHz, Linux 2.6.38

Library	loop	call	cps-call	switch	cond	spawn
nptl	58	63		6 310	63 305	933 689
CPC	58	63	2 018	1 482	3 268	8 519

MIPS-32: MIPS 4KEc at 184 MHz, Linux 2.6.37

All times are in nanoseconds per loop iteration, averaged over millions of runs (smaller is better).
The columns are as follows:
loop: time per loop iteration (reference);
call: time per loop iteration calling a direct-style function;
cps-call: time per loop iteration calling a CPS-converted function;
switch: context switch;
cond: context switch on a condition variable;
spawn: thread creation.

Fig. 5 Speed of thread primitives on various architectures

Function calls Modern processors include specialised hardware for optimising function calls; obviously, this hardware is not able to optimise CPS-converted function calls, which are therefore dramatically slower than direct-style calls. On our very simple benchmark (two embedded loops containing a function call), the *Core 2 Duo* was able to completely

[a] *Pth* never completed, the given value is an educated guess.

[b] On x86-64, performing a direct-style function call adds no measurable delay.

[c] The behaviour of the system call `sched_yield` has changed in Linux 2.6.23, where it has the side-effect of reducing the priority of the calling thread. Until Linux 2.6.38, setting the kernel variable `sched_compat_yield` restored the previous behaviour, which we have done in our x86-32 and MIPS-32 benchmarks. This knob having been removed in Linux 2.6.39, the *nptl* results on x86-64 are highly suspect.

hide the cost of a trivial function call[6], while the function call costs three processor cycles on the *Pentium-M* and just one on the MIPS processor; we do not claim to understand the hardware-level magic that makes such results possible.

With CPS-converted calls, on the other hand, our benchmarking loop takes on the order of 100 processor cycles per iteration (400 on MIPS). This apparently poor result is mitigated by the fact that after goto elimination and lambda-lifting, the loop consists of four CPS function calls; hence, on the *Pentium-M*, a CPS function call is just 6 times slower than a direct-style call, a surprisingly good result considering that our continuations are allocated on the heap [34]. On the MIPS chip the slowdown is closer to a factor of 100, more in line with what we expected.

Context switches and thread creation The "real" primitives are pleasantly fast in CPC. Context switches have similar cost to function calls, which is not surprising since they compile to essentially the same code. While the benchmarks make them appear to be much faster than in any other implementation, including *ST*, this result is skewed by a flaw in the *ST* library, which goes through the event loop on every `st_yield` operation. Context switches on a condition variable are roughly two times slower than pure context switches, which yields timings comparable to those of *ST*.

Thread creation is similarly fast, since it consists in simply allocating a new continuation and registering it with the event loop. This yields results that are ten times faster than *ST*, which must allocate a new stack, and a hundred times faster than the kernel implementation.

4.2 Macro-benchmarks

As we have seen above, the speed of CPC primitives ranges from similar to dramatically faster than that of their equivalents in traditional implementations. However, a CPC program incurs the overhead of the CPS translation, which introduces CPS function calls which, as we have seen above, are much slower than their direct-style equivalents. As it is difficult to predict the number of CPS-converted calls in a CPC program, it is not clear whether this performance increase will be reflected in realistic programs.

We have written a set of naive web servers (less than 200 lines each) that share the exact same structure: a single thread or process waits for incoming connections, and spawns a new thread or process as soon as a client is accepted (there is no support for HTTP/1.1 persistent connections). The servers were benchmarked by repeatedly downloading a tiny file over a dedicated Ethernet with different numbers of simultaneous clients and measuring the average response time. Because of the simple structure of the servers, this simple, repeatable benchmark measures the underlying implementation of concurrency rather than the performance tweaks of a production web server.

Figure 6 presents the results of our experiment. It shows the average serving time per request against the number of concurrent requests; a smaller slope indicates a faster server. It shows that on this particular benchmark all of our web servers show the expected linear behaviour, and that CPC's performance is comparable to that of *ST*, and much better than that of the other implementations. A more in-depth analysis of this benchmark is available in a technical report [26].

[6] Disassembly of the binary shows that the function call is present.

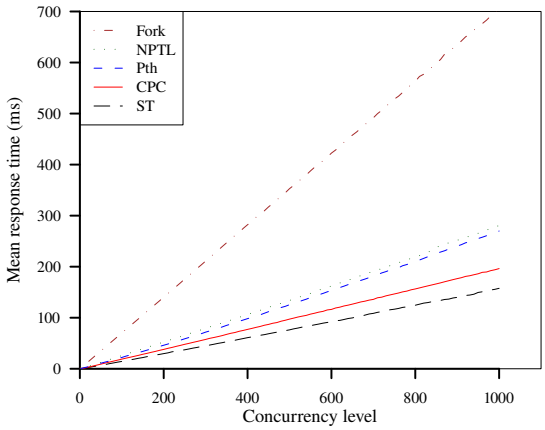

Fig. 6 Web servers comparison

4.3 Hekate

Benchmarking a BitTorrent seeder is a difficult task because it relies either on a real-world load, which is hard to control and only provides seeder-side information, or on an artificial testbed, which might fail to accurately reproduce real-world behaviour. We have however been able to collect enough empirical evidence to show that Hekate (Section 3) is an efficient implementation of BitTorrent, lightweight enough to be run on embedded hardware.

Real-world workload To benchmark the ability of Hekate to sustain a real-world load, we need popular torrents with many requesting peers over a long period of time. Updates for Blizzard's game *World of Warcraft* (WoW), distributed over BitTorrent, meet these conditions: each of the millions of WoW players around the world runs a hidden BitTorrent client, and at any time many of them are looking for the latest update.

We have run an instance of Hekate seeding WoW updates without interruption for weeks. We saw up to 1,000 connected peers (800 on average) and a throughput of up to 10 MB/s (around 5 MB/s on average). Hekate never used more than 10 % of the 3.16 GHz dual core CPU of our benchmarking machine.

Stress-test on embedded hardware We have ported Hekate to *OpenWrt*[7], a Linux distribution for embedded devices. Hekate runs flawlessly on a cheap home router with a 266 MHz *MIPS 4Kc* CPU, 32 MB of RAM and a 100 Mbps network card. The torrent files were kept on a USB flash drive.

Our stress-test consists in 1,000 simultaneous clients (implemented as a single CPC program running on a fast computer directly connected to the device over 100 Mbps Ethernet) requesting random chunks of a 1.2 GB torrent. In these conditions, Hekate was able to serve data at a rate of 2.9 MB/s. The CPU load was pegged at 100%, with most of the CPU time spent servicing software interrupt requests, notably from the USB subsystem (60 % *sirq*, the usb-storage kernel module using up to 25 % of CPU). These data indicate that even on a slow MIPS processor, Hekate's performance is fairly good, and that the performance could be much improved by using a device on which mass storage traffic doesn't go over the USB bus.

[7] `http://openwrt.org`

5 The CPC compilation technique

The current implementation of CPC is structured into three parts: the CPC to C translator, implemented in Objective Caml [32] on top of the CIL framework [35], the runtime, implemented in C, and the standard library, implemented in CPC. The three parts are as independent as possible, and interact only through a small set of well-defined interfaces. This makes it easier to experiment with different approaches.

In this section, we present how the CPC translator turns a CPC program in threaded style into an equivalent C program written in continuation-passing style. Therefore, we only need to focus on the transformations applied to cps functions, ignoring completely the notion of CPC thread or CPC primitive which are handled in the runtime part of CPC. We detail each step of the CPC compilation technique and the difficulties specifically related to the C language.

5.1 Translation passes

The CPC translator is structured in a series of source-to-source transformations which turn a threaded-style CPC program into an equivalent event-driven C program. This sequence of transformations consists of the following passes:

Boxing a small number of variables needs to be encapsulated in environments to ensure the correctness of the later passes;

Splitting the flow of control of each cps function is split into a set of mutually recursive, tail-called, inner functions;

Lambda-lifting free local variables are copied from one inner function to the next, yielding closed inner functions;

CPS conversion at this point, the program is in *CPS-convertible form*, a form simple enough to perform a one-pass partial conversion into continuation-passing style; the resulting continuations are used at runtime by the CPC scheduler.

The converted program is then compiled by a standard C compiler and linked to the CPC scheduler to produce the final executable.

All of these passes are well-known compilation techniques for functional programming languages, but lambda-lifting and CPS conversion are not correct in general for an imperative call-by-value language such as C. The problem is that these transformations copy free variables: if the original variable is modified, the copy becomes out-of-date and the program yields a different result.

Copying is not the only way to handle free variables. When applying lambda-lifting to a call-by-value language with mutable variables, the common solution is to box mutated variables, storing them in environments. However, this is less efficient: adding a layer of indirection hinders cache efficiency and breaks a number of compiler optimisations. Therefore, CPC strives to avoid boxing as much as possible.

One key point of the efficiency of CPC is that we need not box every mutated variable for lambda-lifting and CPS conversion to be correct. As we show in Section 6, even though C is an imperative language, most of the time it is correct to lift functions called in tail position; as it turns out, functions issued from the splitting pass are always called in tail position and it is therefore correct to lift them.

Only a small number of variables, whose addresses have been retained with the “address of” operator (`&`), need to be boxed: we call them extruded variables. Our experiments with

Hekate show that 50 % of local variables in cps functions need to be lifted. Of that number, only 10 % are extruded. In other words, in the largest program written in CPC so far, our translator manages to box only 5 % of the local variables in cps functions.

Sections 5.2 to 5.5 present each pass, explain how they interact and why they are correct. The correctness of the lambda-lifting pass depends on a theorem that will be shown in Section 6.

5.2 CPS conversion

Conversion into Continuation Passing Style [46, 37], or *CPS conversion* for short, is a program transformation technique that makes the flow of control of a program explicit and provides first-class abstractions for it.

Intuitively, the *continuation* of a fragment of code is an abstraction of the action to perform after its execution. For example, consider the following computation:

```
f(g(5) + 4);
```

The continuation of `g(5)` is `f(· + 4)` because the return value of `g` will be added to 4 and then passed to `f`.

CPS conversion consists in replacing every function f in a program with a function f^* taking an extra argument, its *continuation*. Where f would return with value v, f^* invokes its continuation with the argument v. A CPS-converted function therefore never returns, but makes a call to its continuation. Since all of these calls are in tail position, a converted program doesn't use the native call stack: the information that would normally be in the call stack (the dynamic chain) is encoded within the continuation.

This translation has three well-known interesting properties, on which we rely in the implementation of CPC:

CPS conversion need not be global The CPS conversion is not an "all or nothing" deal, in which the complete program must be converted: there is nothing preventing a converted function from calling a function that has not been converted. On the other hand, a function that has not been converted cannot call a function that has, as it does not have a handle to its own continuation.

It is therefore possible to perform CPS conversion on just a subset of the functions constituting a program (in the case of CPC, such functions are annotated with the `cps` keyword). This allows cps code to call code in direct style, for example system calls or standard library functions. Additionally, at least in the case of CPC, a cps function call is much slower than a direct function call; being able to only convert the functions that need the full flexibility of continuations avoids this overhead as much as possible.

Continuations are abstract data structures The classical definition of CPS conversion implements continuations as functions. However, continuations are abstract data structures and functions are only one particular concrete representation of them.

The CPS conversion process performs only two operations on continuations: calling a continuation, which we call *invoke*, and prepending a function application to the body of a continuation, which we call *push*. This property is not really surprising: as continuations are merely a representation for the dynamic chain, it is natural that there should be a correspondence between the operations available on a continuation and a stack.

Since C doesn't have full first-class functions (closures), CPC uses this property to implement continuations as arrays of function pointers and parameters.

Continuation transformers are linear CPS conversion introduces linear (or "one-shot") continuations [5, 7]: when a CPS-converted function receives a continuation, it will use it exactly once, and never duplicate or discard it.

This property is essential for memory management in CPC: as CPC uses the C allocator (`malloc` and `free`) rather than a garbage collector for managing continuations, it allows reliably reclaiming continuations without the need for costly devices such as reference counting.

CPS conversion is not defined for every C function; instead, we restrict ourselves to a subset of functions, which we call the *CPS-convertible* subset. As we shall see in Section 5.4, every C function can be converted to an equivalent function in CPS-convertible form.

The CPS-convertible form restricts the calls to cps functions to make it straightforward to capture their continuation. In CPS-convertible form, a call to a cps function `f` is either in tail position, or followed by a tail call to another cps function whose parameters are *non-shared* variables, that cannot be modified by `f`. This restriction about shared variables is ensured by the *boxing* pass detailed in Section 5.3.

Definition 1 (Extruded and shared variables) *Extruded variables* are local variables (or function parameters) the address of which has been retained using the "address of" operator (`&`).

Shared variables are either extruded or global variables. □

Thus, the set of shared variables includes every variable that might be modified by several functions called in the same dynamic chain.

Definition 2 (CPS-convertible form) A function `h` is in *CPS-convertible form* if every call to a cps function that it contains matches one of the following patterns, where both `f` and `g` are cps functions, a_1, ..., a_n are any C expressions and `x`, y_1, ..., y_n are non-shared variables:

```
                          return f(a1, ..., an);          (1)
x = f(a1, ..., an); return g(x, y1, ..., yn);            (2)
    f(a1, ..., an); return g(x, y1, ..., yn);             (3)
                        f(a1, ..., an); return;           (4)
   f(a1, ..., an); g(x, y1, ..., yn); return;             (5)
x = f(a1, ..., an); g(x, y1, ..., yn); return;            (6)
```

□

We use `return` to explicitly mark calls in tail position. The forms (3) to (6) are only necessary to handle the cases where `f` and `g` return `void`; in the rest of this paper, we ignore these cases and focus on the essential cases (1) and (2).

Early evaluation of non-shared variables To understand why the definition of CPS-convertible form requires non-shared variables for the parameters of `g`, consider what happens when converting a CPS-convertible function. We use a partial CPS conversion, as explained above, focused on tail positions. In a functional language, we would define the CPS conversion as

follows, where $f^\star$ is the CPS-converted form of `f` and `k` the current continuation:

```
                return a;   ⇝  return k(a);
   return f(a1, ..., an);   ⇝  return f*(a1, ..., an, k);
      x = f(a1, ..., an);   ⇝  return f*(a1, ..., an,
return g(x, y1, ..., yn);          λx. g*(x, y1, ..., yn, k));
```

Note the use of the lambda-abstraction λ`x. g`*`(x, y1, ..., yn, k)` to represent the continuation of the call to $f^\star$ in the last case. In a call-by-value language, this continuation can be described as: *"get the return value of* $f^\star$*, evaluate the variables* y_1 *to* y_n *and call* $g^\star$ *with these parameters."*

Representing this continuation in C raises the problem of evaluating the values of y_1 to y_n after the call to $f^\star$ has completed: these variables are local to the CPS-converted function `h` and, as such, allocated in a stack frame which will be discarded when `h` returns. To keep them available until the evaluation of the continuation, we would need to store them in an environment and garbage-collect them at some point. We want to avoid this approach as much as possible for performance reasons since it implies an extra layer of indirection and extra bookkeeping.

We use instead a technique that we call *early evaluation* of variables. It is based on the following property: if we can ensure that the variables y_i cannot be modified by the function `f`, then it is correct to commute their evaluation with the call to `f`. Because the CPC translator produces code where these variables are not shared, thanks to the boxing pass (Section 5.3), it is indeed guaranteed that they cannot be modified by a call to another function in the same dynamic chain. In CPC, the CPS conversion transformation therefore evaluates these variables when the continuation is built, before calling `f`, and stores their values directly in the continuation.

We finally define the CPS conversion pass, using early evaluation and the fact mentioned above that continuations are abstract data structures manipulated by two operators: `push`, which adds a function to the continuation, and `invoke` which executes a continuation.

Definition 3 (CPS conversion) The *CPS conversion* translates the tail positions of every CPS-convertible term as follows, where $f^\star$ is the CPS-converted form of `f` and `k` is the current continuation:

```
                return a;   ⇝  return invoke(k, a);
   return f(a1, ..., an);   ⇝  push (k, f*, a1, ..., an);
                               return invoke(k);
      x = f(a1, ..., an);   ⇝  push (k, g*, y1, ..., yn);
return g(x, y1, ..., yn);      push (k, f*, a1, ..., an);
                               return invoke(k);
```

□

A proof of correctness of this conversion, including a proof that early evaluation is correct in the absence of shared variables, is available in the technical report [28].

Implementation From an implementation standpoint, the continuation `k` is a stack of function pointers and parameters, and `push` adds elements to this stack. The function `invoke` calls the first function of the stack, with the rest of the stack as its parameters. The form `invoke(k, a)` also takes care of passing the value `a` to the continuation `k`; it simply pushes `a` at the top of the continuation before calling its first function.

The current implementation of `push` grows continuations by a multiplicative factor when the array is too small to contain the pushed parameters, and never shrinks them. While this might in principle waste memory in the case of many long-lived continuations with an occasional deep call stack, we believe that this case is rare enough not to justify complicating the implementation with this optimisation.

The function `push` does not align parameters on word boundaries, which leads to smaller continuations and easier store and load operations. Although word-aligned reads and writes are more efficient in general, our tests showed little or no impact in the CPC programs we experimented with, on x86 and x86-64 architectures: the worst case has been a 10 % slowdown in a micro-benchmark with deliberately misaligned parameters. We have reconsidered this trade-off when porting CPC to MIPS, an architecture with no hardware support for unaligned accesses, and added a compilation option to align continuation frames. However we keep this option disabled by default, until we have more experimental data to assess its efficiency.

There is one difference between Definition 3 and the implementation. In a language with proper tail calls, each function would simply invoke the continuation directly; in C, which is not properly tail-recursive, doing that leads to unbounded growth of the native call stack. Therefore, the tail call `return invoke(k)` cannot be used; we work around this issue by using a "trampolining" technique [19]. Instead of calling `invoke`, each cps function returns its continuation: `return k`. The main event loop iteratively receives these continuations and invokes them until a CPC primitive returns `NULL`, which yields to the next CPC thread.

In the following sections, we show how the boxing, splitting and lambda-lifting passes translate any CPC program into CPS-convertible form.

5.3 Boxing

Boxing is a common, straightforward technique to encapsulate mutable variables. It is necessary to ensure the correctness of the CPS conversion and lambda-lifting passes (Sections 5.2 and 5.5). However, boxing induces an expensive indirection to access boxed variables. In order to keep this cost at an acceptably low leve, we box only a subset of all variables — namely extruded variables, whose address is retained with the "address of" operator `&` (see Definition 1, p. 15).

Example Consider the following function:

```
cps void f(int x) {
  int y = 0;
  int *p1 = &x, *p2 = &y;
  /* ... */
  return;
}
```

The local variables `x` and `y` are extruded, because their address is stored in the pointers `p1` and `p2`. The boxing pass allocates them on the heap at the beginning of the function `f` and frees them before `f` returns; every occurrence of `x` is replaced by `*px`, and `&x` by `px`:

```
cps void f(int x) {
  int *px = malloc(sizeof(int));
  int *py = malloc(sizeof(int));
  *px = x;                  /* Initialise px */
  *py = 0;
  int *p1 = px, *p2 = py;
  /* ... with x and y replaced accordingly */
  free(px); free(py);
  return;
}
```

The extruded variables `x` and `y` are not used anymore (except to initialise `px`). Instead, `px` and `py` are used; note that these variables, contrary to `x` and `y`, are not extruded: they hold the address of other variables, but their own address is not retained. After the boxing pass, there are no more extruded variables used in cps functions.

Cost analysis The efficiency of CPC relies in great part on avoiding boxing as much as possible. Performance-wise, we expect boxing only extruded variables to be far less expensive than boxing every lifted variable. Indeed, in a typical C program, few local variables have their address retained compared to the total number of variables.

Experimental data confirm this intuition: in Hekate, the CPC translator boxes 13 variables out of 125 lifted parameters. This result is obtained when compiling Hekate with the current CPC implementation. They take into account the fact that the CPC translator tries to be smart about which variables should be boxed or lifted: for instance, if the address of a variable is retained with the "address of" operator `&` but never used, this variable is not considered as extruded. Using a naive implementation, however, does not change the proportion of boxed variables: with optimisations disabled, 29 variables are boxed out of 323 lifted parameters. In both cases, CPC boxes about 10 % of the lifted variables, which is an acceptable overhead.

Interaction with other passes The boxing pass yields a program without the "address of" operator (`&`); extruded variables are allocated on the heap, and only pointers to them are copied by the lambda-lifting and CPS-conversion passes rather than extruded variables themselves. One may wonder, however, whether it is correct to perform boxing before every other transformation. It turns out that boxing does not interfere with the other passes, because they do not introduce any additional "address of" operators. The program therefore remains free of extruded variables. Moreover, it is preferable to box early, before introducing inner functions, since it makes it easier to identify the entry and exit points of the original function, where variables are allocated and freed.

Extruded variables and tail calls Although we keep the cost of boxing low, with about 10 % of boxed variables, boxing has another, hidden cost: it breaks tail recursive cps calls. Since the boxed variables might, in principle, be used during the recursive calls, one cannot free them beforehand. Therefore, functions featuring extruded variables do not benefit from the automatic elimination of tail recursive calls induced by the CPS conversion. While this prevents CPC from optimising tail recursive calls "for free", it is not a real limitation: the C language does not ensure the elimination of tail recursive calls anyway, as the stack frame should similarly be preserved in case of extruded variables, and C programmers are used not to rely on it.

5.4 Splitting

To transform a CPC program into CPS-convertible form, the CPC translator needs to ensure that every call to a cps function is either in tail position or followed by a tail call to another cps function. In the original CPC code, calls to cps functions might, by chance, respect this property but, more often than not, they are followed by some direct-style code. The role of the CPC translator is therefore to replace this direct-style chunk of code following a cps call by a call to a cps function which encapsulates this chunk. We call this pass *splitting* because it splits each original cps function into many, mutually recursive, cps functions in CPS-convertible form.

The splitting pass is performed in two steps. The first steps consists in adding `goto` statements, in particular after cps function calls which are not in tail position. It produces code with an explicit flow of control, which resembles CPS-convertible form except for some `goto` instead of tail calls (Section 5.4.1). The second step then converts these `goto` statements into calls to inner cps functions, yielding to CPS-convertible form (Section 5.4.2).

5.4.1 Explicit flow of control

When a cps call is not in tail position, or followed by a tail cps call, the CPC translator adds a `goto` statement after it. This `goto` is to be converted into a tail cps call in the next step.

The simplest case is trivial:

```
cpc_yield(); rc = 0;
```

becomes

```
cpc_yield(); goto l;
l: rc = 0;
```

Loops and conditional jumps require more care:

```
while(!timeout) {
  int rc = cpc_read();
  if(rc <= 0) break;
  cpc_write();
}
reset_timeout();
```

is converted to:

```
while_label:
  if(!timeout) {
    int rc = cpc_read(); goto l;
    l:
      if(rc <= 0) goto break_label;
      cpc_write(); goto while_label;
  }
break_label:
  reset_timeout();
```

More generally, when the flow of control after a cps call goes outside of a loop (`for`, `while` or `do ... while`) or a `switch` statement, the CPC translator simplifies these constructs into `if` and `goto` statements, adding the necessary labels on the fly.

Although these transformations bring the code in a shape close to CPS-convertible form, we need to add some more `goto` statements for the second step to correctly encapsulate chunks of code into cps functions. Consider for instance the following piece of code:

```
if(rc < 0) {
  cpc_yield(); rc = 0;
}
printf("rc = %d\n", rc);
return rc;
```

With the rules described above, it would be translated to:

```
if(rc < 0) {
  cpc_yield(); goto l;
  l: { rc = 0; }
}
printf("rc = %d\n", rc);
return rc;
```

This is not enough because the block labelled by `l` will be converted to a cps function. But if we encapsulate only the {`rc = 0;`} part, we will miss the call to `printf` when `rc < 0`. Therefore, to ensure a correct encapsulation in the second step, we also need to make the flow of control explicit when exiting a labelled block. The example becomes:

```
if(rc < 0) {
  cpc_yield(); goto l;
  l: { rc = 0; goto done; }
}
done:
  printf("rc = %d\n", rc);
  return rc;
```

After this step, every cps call is either in tail position or followed by a `goto statement`, and every labelled block exits with either a `return` or a `goto` statement.

5.4.2 Eliminating gotos

It is a well-known fact in the compiler folklore that a tail call is equivalent to a goto. It is perhaps less known that a goto is equivalent to a tail call [45, 54]: the block of any destination label `l` is encapsulated in an inner function `l()`, and each `goto l;` is replaced by a tail call to `l`.

Coming back to the first example of Section 5.4.1, the label `l` yields a function `l()`:

```
f(); return l();
cps void l() { rc = 0; }
```

We see that the first line is now in CPS-convertible form. Note again that we use `return` to mark tail calls explicitly in the C code.

Applying the same conversion to loops yields mutually recursive functions. For instance, the `while` loop in the second example is converted into `while_label()`, `l()` and `break_label()`:

```
while_label();
cps void while_label() {
  if(!timeout) {
    int rc = cpc_read(); return l();
    cps void l() {
      if(rc <= 0) return break_label();
      cpc_write(); return while_label();
    }
  }
}
cps void break_label() {
  reset_timeout();
}
```

When a label, like `while_label` above, is reachable not only through a `goto`, but also directly through the linear flow of the program, it is necessary to call its associated function at the point of definition; this is what we do on the first line of this example.

Note that splitting may introduce free variables; for instance, in the previous example, `rc` is free in the function `l`. In this intermediate C-like code, inner functions are considered just like any other C block when it comes to the scope and lifespan of variables: each variable lives as long as the block or function which defines it, and it can be read and modified by any instruction inside this block, including inner functions. There is in particular no copy of free variables in inner functions; variables are shared inside their block. In the previous example, the lifespan of the variable `rc` is the `if` block, including the call to the function `l` which reads the free variable allocated in its enclosing function `while_label`.

The third example shows the importance of having explicit flow of control at the end of labelled blocks. After `goto` elimination, it becomes:

```
if(rc < 0) {
  cpc_yield(); return l();
  cps int l() { rc = 0; return done(); }
}
cps int done() {
  printf("rc = %d\n", rc);
  return rc;
}
return done();
```

Note the tail call to `done` at the end of `l` to ensure that the `printf` is executed when `rc < 0`, and again the call on the last line to execute it otherwise.

5.4.3 Implementation

The actual implementation of the CPC translator implements splitting on an as-needed basis: a given subtree of the abstract syntax tree (AST) is only transformed if it contains CPC primitives that cannot be implemented in direct style. Our main concern here is to transform the code as little as possible, on the assumption that the gcc compiler is optimised for human-written code.

To perform splitting, the CPC translator iterates over the AST, checking whether the cps functions are in CPS-convertible form and interleaving the two steps described above

to reach CPS-convertible form incrementally. On each pass, when the translator finds a cps call it dispatches on the statement following it:

- in the case of a `return`, the fragment is already CPS-convertible and the translator continues;
- in the case of a `goto`, it is converted into a tail call, with the corresponding label turned into an inner function (Section 5.4.2), and the translator starts another pass;
- for any other statement, a `goto` is added to make the flow of control explicit, converting enclosing loops too if necessary (Section 5.4.1). The translator then starts another pass and will eventually convert the introduced `goto` into a tail call.

At the end of the splitting pass, the translated program is in CPS-convertible form. However, it is not quite ready for CPS conversion because we introduced inner functions, which makes it invalid C. In particular, these functions may contain free variables. The next pass, lambda-lifting, takes care of these free variables to get a valid C program in CPS-convertible form, suitable for the CPS conversion pass described in Section 5.2.

5.5 Lambda-lifting

Lambda-lifting [25] is a standard technique to eliminate free variables in functional languages. It proceeds in two phases [11]. In the first pass ("parameter lifting"), free variables are replaced by local, bound variables. In the second pass ("block floating"), the resulting closed functions are floated to top-level.

Example Coming back to the latest example, we add an enclosing function `f` to define `rc` and make the fragment self-contained:

```
cps int f(int rc) {
  if(rc < 0) {
    cpc_yield(); return l();
    cps int l() { rc = 0; return done(); }
  }
  cps int done() {
    printf("rc = %d\n", rc);
    return rc;
  }
  return done();
}
```

The function `f` contains two inner functions, `l` and `done`. The local variable `rc` is used as a free variable in both of these functions.

Parameter lifting consists in adding the free variable as a parameter of every inner function:

```
cps int f(int rc) {
  if(rc < 0) {
    cpc_yield(); return l(rc);
    cps int l(int rc) { rc = 0; return done(rc); }
  }
  cps int done(int rc) {
```

```
    printf("rc = %d\n", rc);
    return rc;
  }
  return done(rc);
}
```

Note that `rc` is now a parameter of `l` and `done`, and has been added accordingly whenever these functions where called. There are, now, three copies of `rc`; alpha-conversion makes this more obvious:

```
cps int f(int rc1) {
  if(rc1 < 0) {
    cpc_yield(); return l(rc1);
    cps int l(int rc2) { rc = 0; return done(rc2); }
  }
  cps int done(int rc3) {
    printf("rc = %d\n", rc3);
    return rc3;
  }
  return done(rc1);
}
```

Once the parameter lifting step has been performed, there is no free variable anymore and the block floating step consists in extracting these inner, closed functions at top-level:

```
cps int f(int rc1) {
  if(rc1 < 0) {
    cpc_yield(); return l(rc1);
  }
  return done(rc1);
}
cps int l(int rc2) {
  rc = 0;
  return done(rc2);
}
cps int done(int rc3) {
  printf("rc = %d\n", rc3);
  return rc3;
}
```

Applying boxing, splitting and lambda-lifting always yields CPS-convertible programs:

- every call to a cps function is either a tail call (not affected by the transformations) or followed by a tail cps call (introduced in the splitting pass),
- the parameters of this second cps call are local variables, since they are introduced by the lambda-lifting pass,
- these parameters are not shared because they are neither global (local variables) nor extruded (the boxing pass ensures that there are no more extruded variables in the program).

Lambda-lifting in imperative languages There is one major issue with applying lambda-lifting to C extended with inner functions: this transformation is in general not correct in a call-by-value languages with mutable variables.

Consider what would happen if splitting were to yield the following code:

```
cps int f(int rc) {
  cps void set() { rc = 0; return; }
  cps void done() {
    printf("rc = %d\n", rc);
    return;
  }
  set(); return done();
}
```

This code, which is in CPS-convertible form, prints out `rc = 0` whatever the original value of `rc` was: the call to `set` sets `rc` to 0 and the call to `done` displays it.

Once lambda-lifted, this code becomes:

```
cps int f(int rc1) {
  set(rc1); return done(rc1);
}
cps void set(int rc2) {
  rc2 = 0;
  return;
}
cps void done(int rc3) {
  printf("rc = %d\n", rc3);
  return;
}
```

The result has changed: the code now displays the original value of `rc` passed to `f` rather than 0. The reason why lambda-lifting is incorrect in that case is because `set` and `done` work on two separate copies `rc`, `rc2` and `rc3`, whereas in the original code there was only one instance of `rc` shared by all inner functions.

This issue is triggered by the fact that the function `set` is not called in tail position. This non-tail call allows us to observe the incorrect value of `rc3` after `set` has modified the copy `rc2` and returned. If `set` were called in tail position instead, the fact that it operates on a copy of `rc1` would remain unnoticed.

Lambda-lifting and tail calls In fact, although lambda-lifting is not correct in general in a call-by-value language with mutable variables, it becomes correct once restricted to functions called in tail position and non-extruded parameters. More precisely, in the absence of extruded variables, it is safe to lift a parameter provided every inner function where this parameter is lifted is only called in tail position. We show this result in Section 6 (Theorem 1, p. 29).

Inner functions in CPC are the result of `goto` elimination during the splitting step. As a result, they are always called in tail position. Moreover, as explained in Section 5.3, the boxing pass ensures that extruded variables have been eliminated at this point of the transformation. Hence, lambda-lifting is correct in the case of CPC.

Implementation To lift as few variables as possible, lambda-lifting is implemented incrementally. Rather than lifting every parameter, the translators looks for free variables to be lifted in a cps function and adds them as parameters at call points; this creates new free variables, and the translator iterates until it reaches a fixed point.

This implementation might be further optimised with a liveness analysis, which would in particular avoid lifting uninitialised parameters. The current translator performs a very limited analysis: only variables which are used (hence alive) in a single function are not lifted.

Experimental results The common technique to use lambda-lifting in an imperative language is to box every mutated variable, in order to duplicate pointers to these variables instead of the variables themselves. To quantify the amount of boxing avoided by our technique of lambda-lifting tail-called functions, we used a modified version of CPC which blindly boxes every lifted parameter and measure the amount of boxing that it induced in Hekate, the most substantial program written with CPC so far.

Hekate contains 260 local variables and function parameters, spread across 28 cps functions[8]. Among them, 125 variables are lifted. A naive lambda-lifting pass would therefore need to box almost 50 % of the variables.

On the other hand, boxing extruded variables only carries a much smaller overhead: in Hekate, the current CPC translator boxes only 5 % of the variables in cps functions. In other words, 90 % of the lifted variables in Hekate are safely left unboxed, keeping the overhead associated with boxing to a reasonable level.

6 Lambda-lifting in an imperative language

To prove the correctness of lambda-lifting in an imperative, call-by-value language when functions are called in tail position, we do not reason directly on CPC programs, because the semantics of C is too broad and complex for our purposes. The CPC translator leaves most parts of converted programs intact, transforming only control structures and function calls. Therefore, we define a simple language with restricted values, expressions and terms, that captures the features we are most interested in (Section 6.1).

The reduction rules for this language (Section 6.1.1) use a simplified memory model without pointers and enforce that local variables are not accessed outside of their scope, as ensured by our boxing pass. This is necessary since we have shown that lambda-lifting is not correct in general in the presence of extruded variables.

It turns out that the “naive” reduction rules defined in Section 6.1.1 do not provide strong enough invariants to prove this correctness theorem by induction, mostly because we represent memory with a store that is deeply modified by lambda-lifting. Therefore, in Section 6.2, we define an equivalent, “optimised” set of reduction rules which enforces more regular stores and closures.

The proof of correctness is then carried in Section 6.3 using these optimised rules. We first define the invariants needed for the proof and formulate a strengthened version of the correctness theorem (Theorem 3, Section 6.3.1). A comprehensive overview of the proof is then given in Section 6.3.2. The proof is fully detailed in Section 6.3.5, with the help of a few lemmas to keep the main proof shorter (Sections 6.3.3 and 6.3.4).

The main limitation of this proof is that Theorems 1 and 3 are implications, not equivalences: we do not prove that if a term does not reduce, it will not reduce once lifted. For instance, this proof does not ensure that lambda-lifting does not break infinite loops.

[8] These numbers leave out direct-style functions, which do not need to be converted, and around 200 unused temporary variables introduced by a single pathological macro-expansion in the *curl* library.

6.1 Definitions

In this section, we define the terms (Definition 4), the reduction rules (Section 6.1.1) and the lambda-lifting transformation itself (Section 6.1.2) for our small imperative language. With these preliminary definitions, we are then able to characterise *liftable parameters* (Definition 11) and state the main correctness theorem (Theorem 1, Section 6.1.3).

Definition 4 (Values, expression and terms)
Values are either boolean and integer constants or **1**, a special value for functions returning `void`.

$$v ::= \quad \mathbf{1} \mid \mathbf{true} \mid \mathbf{false} \mid n \in \mathbf{N}$$

Expressions are either values or variables. We deliberately omit arithmetic and boolean operators, with the sole concern of avoiding boring cases in the proofs.

$$expr ::= \quad v \mid x \mid \ldots$$

Terms are made of assignments, conditionals, sequences, recursive functions definitions and calls.

$$\begin{aligned} T ::= & expr \mid x := T \mid \mathbf{if}\ T\ \mathbf{then}\ T\ \mathbf{else}\ T \mid T\ ;\ T \\ & \mid \mathbf{letrec}\ f(x_1 \ldots x_n) = T\ \mathbf{in}\ T \mid f(T, \ldots, T) \end{aligned}$$

□

Our language focuses on the essential details affected by the transformations: recursive functions, conditionals and memory accesses. Loops, for instance, are ignored because they can be expressed in terms of recursive calls and conditional jumps — and that is, in fact, how the splitting pass translates them (Section 5.4). Since lambda-lifting happens after the splitting pass, our language need to include inner functions (although they are not part of the C language), but it can safely exclude `goto` statements.

6.1.1 Naive reduction rules

Environments and stores Handling inner functions requires explicit closures in the reduction rules. We need environments, written ρ, to bind variables to locations, and a store, written s, to bind locations to values.

Environments and *stores* are partial functions, equipped with a single operator which extends and modifies a partial function: $\cdot + \{\cdot \mapsto \cdot\}$.

Definition 5 The modification (or extension) f' of a partial function f, written $f' = f + \{x \mapsto y\}$, is defined as follows:

$$f'(t) = \begin{cases} y & \text{when } t = x \\ f(t) & \text{otherwise} \end{cases}$$
$$\mathrm{dom}(f') = \mathrm{dom}(f) \cup \{x\}$$

□

$$(\text{VAL})\ \frac{}{v^s \xrightarrow[\mathscr{F}]{\rho} v^s} \qquad (\text{VAR})\ \frac{\rho\ x = l \in \text{dom}\ s}{x^s \xrightarrow[\mathscr{F}]{\rho} s\ l^s}$$

$$(\text{ASSIGN})\ \frac{a^s \xrightarrow[\mathscr{F}]{\rho} v^{s'} \qquad \rho\ x = l \in \text{dom}\ s'}{x \coloneqq a^s \xrightarrow[\mathscr{F}]{\rho} \mathbf{1}^{s'+\{l \mapsto v\}}} \qquad (\text{SEQ})\ \frac{a^s \xrightarrow[\mathscr{F}]{\rho} v^{s'} \qquad b^{s'} \xrightarrow[\mathscr{F}]{\rho} v'^{s''}}{a\ ;\ b^s \xrightarrow[\mathscr{F}]{\rho} v'^{s''}}$$

$$(\text{IF-T.})\ \frac{a^s \xrightarrow[\mathscr{F}]{\rho} \textbf{true}^{s'} \qquad b^{s'} \xrightarrow[\mathscr{F}]{\rho} v^{s''}}{\textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c^s \xrightarrow[\mathscr{F}]{\rho} v^{s''}} \qquad (\text{IF-F.})\ \frac{a^s \xrightarrow[\mathscr{F}]{\rho} \textbf{false}^{s'} \qquad c^{s'} \xrightarrow[\mathscr{F}]{\rho} v^{s''}}{\textbf{if}\ a\ \textbf{then}\ b\ \textbf{else}\ c^s \xrightarrow[\mathscr{F}]{\rho} v^{s''}}$$

$$(\text{LETREC})\ \frac{b^s \xrightarrow[\mathscr{F}']{\rho} v^{s'} \qquad \mathscr{F}' = \mathscr{F} + \{f \mapsto [\lambda x_1 \ldots x_n.a, \rho, \mathscr{F}]\}}{\textbf{letrec}\ \ f(x_1 \ldots x_n) = a\ \textbf{in}\ b^s \xrightarrow[\mathscr{F}]{\rho} v^{s'}}$$

$$(\text{CALL})\ \frac{\mathscr{F}\ f = \left[\lambda x_1 \ldots x_n.b, \rho', \mathscr{F}'\right] \qquad \rho'' = (x_1, l_1) \cdot \ldots \cdot (x_n, l_n) \qquad l_i \text{ fresh and distinct}}{\forall i, a_i^{s_i} \xrightarrow[\mathscr{F}]{\rho} v_i^{s_{i+1}} \qquad b^{s_{n+1}+\{l_i \mapsto v_i\}} \xrightarrow[\mathscr{F}'+\{f \mapsto \mathscr{F}\ f\}]{\rho''\cdot\rho'} v^{s'}}{f(a_1 \ldots a_n)^{s_1} \xrightarrow[\mathscr{F}]{\rho} v^{s'}}$$

Fig. 7 "Naive" reduction rules

Definition 6 (Environments of variables and functions) Environments of variables are defined inductively by

$$\rho ::= \varepsilon \mid (x, l) \cdot \rho,$$

i.e. the empty domain function and $\rho + \{x \mapsto l\}$ (respectively).

Environments of functions, on the other hand, associate function names to closures:

$$\mathscr{F} : \{f, g, h, \ldots\} \to \{[\lambda x_1 \ldots x_n.T, \rho, \mathscr{F}]\}.$$

□

Note that although we have a notion of locations, which correspond roughly to memory addresses in C, there is no way to copy, change or otherwise manipulate a location directly in the syntax of our language. This is on purpose, since adding this possibility would make lambda-lifting incorrect: it translates the fact, ensured by the boxing pass in the CPC translator, that there are no extruded variables in the lifted terms.

Reduction rules We use classical big-step reduction rules for our language (Figure 7, p. 27).

In the (call) rule, we need to introduce *fresh* locations for the parameters of the called function. This means that we must choose locations that are not already in use, in particular in the environments ρ' and $\mathscr{F}$. To express this choice, we define two ancillary functions, Env and Loc, to extract the environments and locations contained in the closures of a given environment of functions $\mathscr{F}$.

Definition 7 (Set of environments, set of locations)

$$\text{Env}(\mathscr{F}) = \bigcup \left\{\rho, \rho' \mid \left[\lambda x_1 \ldots x_n.M, \rho, \mathscr{F}'\right] \in \text{Im}(\mathscr{F}), \rho' \in \text{Env}(\mathscr{F}')\right\}$$

$$\text{Loc}(\mathscr{F}) = \bigcup \{\text{Im}(\rho) \mid \rho \in \text{Env}(\mathscr{F})\}$$

$$\text{A location } l \text{ is said to } \textit{appear} \text{ in } \mathscr{F} \quad \text{iff} \quad l \in \mathrm{Loc}(\mathscr{F}).$$

□

These functions allow us to define fresh locations.

Definition 8 (Fresh location) In the (call) rule, a location is *fresh* when:

- $l \notin \mathrm{dom}(s_{n+1})$, i.e. l is not already used in the store before the body of f is evaluated, and
- l doesn't appear in $\mathscr{F}' + \{f \mapsto \mathscr{F}\, f\}$, i.e. l will not interfere with locations captured in the environment of functions.

□

Note that the second condition implies in particular that l does not appear in either $\mathscr{F}$ or ρ'.

6.1.2 Lambda-lifting

We mentioned in Section 5.5 that lambda-lifting can be split into two parts: parameter lifting and block floating. We will focus only on the first part here, since the second one is trivial. Parameter lifting consists in adding a free variable as a parameter of every inner function where it appears free. This step is repeated until every variable is bound in every function, and closed functions can safely be floated to top-level. Note that although the transformation is called lambda-lifting, we do not focus on a single function and try to lift all of its free variables; on the contrary, we define the lifting of a single free parameter x in every possible function.

Usually, smart lambda-lifting algorithms strive to minimize the number of lifted variables. Such is not our concern in this proof: parameters are lifted in every function where they might potentially be free. (In our implementation, the CPC translator actually uses a smarter approach to avoid lifting too many parameters, as explained in Section 5.5.)

Definition 9 (Parameter lifting in a term) Assume that x is defined as a parameter of a given function g, and that every inner function in g is called h_i (for some $i \in \mathbf{N}$). Also assume that function parameters are unique before lambda-lifting.
Then, the *lifted form* $(M)_*$ of the term M with respect to x is defined inductively as follows:

$$(\mathbf{1})_* = \mathbf{1} \qquad (n)_* = n$$
$$(true)_* = true \qquad (false)_* = false$$
$$(y)_* = y \quad \text{and} \quad (y := a)_* = y := (a)_* \quad (\text{even if } y = x)$$
$$(a\ ;\ b)_* = (a)_*\ ;\ (b)_*$$
$$(\textbf{if } a \textbf{ then } b \textbf{ else } c)_* = \textbf{if } (a)_* \textbf{ then } (b)_* \textbf{ else } (c)_*$$
$$(\textbf{letrec } f(x_1 \ldots x_n) = a \textbf{ in } b)_* = \begin{cases} \textbf{letrec } f(x_1 \ldots x_n x) = (a)_* \textbf{ in } (b)_* & \text{if } f = h_i \\ \textbf{letrec } f(x_1 \ldots x_n) = (a)_* \textbf{ in } (b)_* & \text{otherwise} \end{cases}$$
$$(f(a_1 \ldots a_n))_* = \begin{cases} f((a_1)_*, \ldots, (a_n)_*, x) & \text{if } f = h_i \text{ for some } i \\ f((a_1)_*, \ldots, (a_n)_*) & \text{otherwise} \end{cases}$$

□

6.1.3 Correctness condition

We claim that parameter lifting is correct for variables defined in functions whose inner functions are called exclusively in *tail position*. We call these variables *liftable parameters*.

We first define tail positions as usual [10]:

Definition 10 (Tail position) *Tail positions* are defined inductively as follows:

1. M and N are in tail position in **if** P **then** M **else** N.
2. N is in tail position in N and M ; N and **letrec** $f(x_1 \ldots x_n) = M$ **in** N.

□

A parameter x defined in a function g is liftable if every inner function in g is called exclusively in tail position.

Definition 11 (Liftable parameter) A parameter x is *liftable* in M when:

- x is defined as the parameter of a function g,
- inner functions in g, named h_i, are called exclusively in tail position in g or in one of the h_i.

□

Our main theorem is that performing parameter-lifting on a liftable parameter preserves the reduction:

Theorem 1 (Correctness of lambda-lifting) *If x is a liftable parameter in M, then*

$$\exists t, M^{\varepsilon} \xrightarrow[\varepsilon]{} v^{t} \text{ implies } \exists t', (M)^{\varepsilon}_{*} \xrightarrow[\varepsilon]{} v^{t'}.$$

Note that the resulting store t' changes because lambda-lifting introduces new variables, hence new locations in the store, and changes the values associated with lifted variables; Section 6.3 is devoted to the proof of this theorem. To maintain invariants during the proof, we need to use an equivalent, "optimised" set of reduction rules; it is introduced in the next section.

6.2 Optimised reduction rules

The naive reduction rules (Section 6.1.1) are not well-suited to prove the correctness of lambda-lifting. Indeed, the proof is by induction and requires a number of invariants on the structure of stores and environments. Rather than having a dozen of lemmas to ensure these invariants during the proof of correctness, we translate them as constraints in the reduction rules.

To this end, we introduce two optimisations — minimal stores (Section 6.2.1) and compact closures (Section 6.2.2) — which lead to the definition of an optimised set of reduction rules (Figure 8, Section 6.2.3). The equivalence between optimised and naive reduction rules is shown in the technical report [28].

6.2.1 Minimal stores

In the naive reduction rules, the store grows faster when reducing lifted terms, because each function call adds to the store as many locations as it has function parameters. This yields stores of different sizes when reducing the original and the lifted term, and that difference cannot be accounted for locally, at the rule level.

Consider for instance the simplest possible case of lambda-lifting:

$$\textbf{letrec } g(x) = (\textbf{letrec } h() = x \textbf{ in } h()) \textbf{ in } g(\mathbf{1}) \qquad \text{(original)}$$

$$\textbf{letrec } g(x) = (\textbf{letrec } h(y) = y \textbf{ in } h(x)) \textbf{ in } g(\mathbf{1}) \qquad \text{(lifted)}$$

At the end of the reduction, the store for the original term is $\{l_x \mapsto \mathbf{1}\}$ whereas the store for the lifted term is $\{l_x \mapsto \mathbf{1}; l_y \mapsto \mathbf{1}\}$. More complex terms would yield even larger stores, with many out-of-date copies of lifted variables.

To keep the store under control, we need to get rid of useless variables as soon as possible during the reduction. It is safe to remove a variable x from the store once we are certain that it will never be used again, i.e. as soon as the term in tail position in the function which defines x has been evaluated. This mechanism is analogous to the deallocation of a stack frame when a function returns.

To track the variables whose location can be safely reclaimed after the reduction of some term M, we introduce *split environments*. Split environments are written $\rho_T|\rho$, where ρ_T is called the *tail environment* and ρ the non-tail one; only the variables belonging to the tail environment may be safely reclaimed. The reduction rules build environments so that a variable x belongs to ρ_T if and only if the term M is in tail position in the current function f and x is a parameter of f. In that case, it is safe to discard the locations associated to all of the parameters of f, including x, after M has been reduced because we are sure that the evaluation of f is completed (and there is no first-class functions in the language to keep references on variables beyond their scope of definition).

We also define a *cleaning* operator, $\cdot \setminus \cdot$, to remove a set of variables from the store.

Definition 12 (Cleaning of a store) The store s cleaned with respect to the variables in ρ, written $s \setminus \rho$, is defined as $s \setminus \rho = s|_{\mathrm{dom}(s)\setminus\mathrm{Im}(\rho)}$. □

6.2.2 Compact closures

Another source of complexity with the naive reduction rules is the inclusion of useless variables in closures. It is safe to remove from the environments of variables contained in closures the variables that are also parameters of the function: when the function is called, and the environment restored, these variables will be hidden by the freshly instantiated parameters.

This is typically what happens to lifted parameters: they are free variables, captured in the closure when the function is defined, but these captured values will never be used since calling the function adds fresh parameters with the same names. We introduce *compact closures* in the optimised reduction rules to avoid dealing with this hiding mechanism in the proof of lambda-lifting.

A compact closure is a closure that does not capture any variable which would be hidden when the closure is called because of function parameters having the same name.

Definition 13 (Compact closure and environment) A closure $[\lambda x_1 \ldots x_n.M, \rho, \mathscr{F}]$ is *compact* if $\forall i, x_i \notin \mathrm{dom}(\rho)$ and $\mathscr{F}$ is compact. An environment is *compact* if it contains only compact closures. □

$$(\text{VAL})\ \frac{}{v^s \xrightarrow[\mathscr{F}]{\rho_T|\rho} v^{s\setminus\rho_T}} \qquad (\text{VAR})\ \frac{\rho_T\cdot\rho\ x = l \in \text{dom}\ s}{x^s \xrightarrow[\mathscr{F}]{\rho_T|\rho} s\ l^{s\setminus\rho_T}}$$

$$(\text{ASSIGN})\ \frac{a^s \xrightarrow[\mathscr{F}]{|\rho_T\cdot\rho} v^{s'} \qquad \rho_T\cdot\rho\ x = l \in \text{dom}\ s'}{x := a^s \xrightarrow[\mathscr{F}]{\rho_T|\rho} \mathbf{1}^{s'+\{l\mapsto v\}\setminus\rho_T}} \qquad (\text{SEQ})\ \frac{a^s \xrightarrow[\mathscr{F}]{|\rho_T\cdot\rho} v^{s'} \qquad b^{s'} \xrightarrow[\mathscr{F}]{\rho_T|\rho} v'^{s''}}{a\ ;\ b^s \xrightarrow[\mathscr{F}]{\rho_T|\rho} v'^{s''}}$$

$$(\text{IF-T.})\ \frac{a^s \xrightarrow[\mathscr{F}]{|\rho_T\cdot\rho} \mathbf{true}^{s'} \qquad b^{s'} \xrightarrow[\mathscr{F}]{\rho_T|\rho} v^{s''}}{\mathbf{if}\ a\ \mathbf{then}\ b\ \mathbf{else}\ c^s \xrightarrow[\mathscr{F}]{\rho_T|\rho} v^{s''}} \qquad (\text{IF-F.})\ \frac{a^s \xrightarrow[\mathscr{F}]{|\rho_T\cdot\rho} \mathbf{false}^{s'} \qquad c^{s'} \xrightarrow[\mathscr{F}]{\rho_T|\rho} v^{s''}}{\mathbf{if}\ a\ \mathbf{then}\ b\ \mathbf{else}\ c^s \xrightarrow[\mathscr{F}]{\rho_T|\rho} v^{s''}}$$

$$(\text{LETREC})\ \frac{b^s \xrightarrow[\mathscr{F}']{\rho_T|\rho} v^{s'} \qquad \rho' = \rho_T\cdot\rho|_{\text{dom}(\rho_T\cdot\rho)\setminus\{x_1\ldots x_n\}} \qquad \mathscr{F}' = \mathscr{F} + \{f \mapsto [\lambda x_1\ldots x_n.a, \rho', \mathscr{F}]\}}{\mathbf{letrec}\ f(x_1\ldots x_n) = a\ \mathbf{in}\ b^s \xrightarrow[\mathscr{F}]{\rho_T|\rho} v^{s'}}$$

$$(\text{CALL})\ \frac{\mathscr{F}\ f = [\lambda x_1\ldots x_n.b, \rho', \mathscr{F}'] \qquad \rho'' = (x_1,l_1)\cdot\ldots\cdot(x_n,l_n) \qquad l_i \text{ fresh and distinct} \qquad \forall i, a_i^{s_i} \xrightarrow[\mathscr{F}]{|\rho_T\cdot\rho} v_i^{s_{i+1}} \qquad b^{s_{n+1}+\{l_i\mapsto v_i\}} \xrightarrow[\mathscr{F}'+\{f\mapsto\mathscr{F}\ f\}]{\rho''|\rho'} v^{s'}}{f(a_1\ldots a_n)^{s_1} \xrightarrow[\mathscr{F}]{\rho_T|\rho} v^{s'\setminus\rho_T}}$$

Fig. 8 Optimised reduction rules

We define a canonical mapping from any environment $\mathscr{F}$ to a compact environment $\mathscr{F}_*$, restricting the domains of every closure in $\mathscr{F}$.

Definition 14 (Canonical compact environment) The *canonical compact environment* $\mathscr{F}_*$ is the unique environment with the same domain as $\mathscr{F}$ such that

$$\begin{aligned}\forall f \in \text{dom}(\mathscr{F}), \mathscr{F}\ f = [\lambda x_1\ldots x_n.M, \rho, \mathscr{F}'] \\ \text{implies } \mathscr{F}_*\ f = [\lambda x_1\ldots x_n.M, \rho|_{\text{dom}(\rho)\setminus\{x_1\ldots x_n\}}, \mathscr{F}'_*]\,.\end{aligned}$$

□

6.2.3 Optimised reduction rules

Combining both optimisations yields the *optimised* reduction rules (Figure 8, p. 31), used Section 6.3 for the proof of lambda-lifting. We ensure minimal stores by cleaning them in the (val), (var) and (assign) rules, which correspond to tail positions; split environments are introduced in the (call) rule to distinguish fresh parameters, to be cleaned, from captured variables, to preserve. Tail positions are tracked in every rule through split environments, to avoid cleaning variables too early, in a non-tail branch.

We also build compact closures in the (letrec) rule by removing the parameters of f from the captured environment ρ'.

Theorem 2 (Equivalence between naive and optimised reduction rules) *Optimised and naive reduction rules are equivalent: every reduction in one set of rules yields the same result in the other. It is necessary, however, to take care of locations left in the store by the naive reduction:*

$$M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon|\varepsilon} v^\varepsilon \quad \textit{iff} \quad \exists s, M^\varepsilon \xrightarrow[\varepsilon]{\varepsilon} v^s$$

The proof of this theorem is detailed in the technical report [28].

6.3 Correctness of lambda-lifting

In this section, we prove the correctness of lambda-lifting (Theorem 1, p. 29) by induction on the height of the optimised reduction.

Section 6.3.1 defines stronger invariants and rewords the correctness theorem with them. Section 6.3.2 gives an overview of the proof. Sections 6.3.3 and 6.3.4 prove a few lemmas needed for the proof. Section 6.3.5 contains the actual proof of correctness.

6.3.1 Strengthened hypotheses

We need strong induction hypotheses to ensure that key invariants about stores and environments hold at every step. For that purpose, we define *aliasing-free environments*, in which locations may not be referenced by more than one variable, and *local positions*. They yield a strengthened version of liftable parameters (Definition 17). We then define lifted environments (Definition 18) to mirror the effect of lambda-lifting in lifted terms captured in closures, and finally reformulate the correctness of lambda-lifting in Theorem 3 with hypotheses strong enough to be provable directly by induction.

Definition 15 (Aliasing) A set of environments $\mathscr{E}$ is *aliasing-free* when:

$$\forall \rho, \rho' \in \mathscr{E}, \forall x \in \mathrm{dom}(\rho), \forall y \in \mathrm{dom}(\rho'),\ \rho\ x = \rho'\ y \Rightarrow x = y.$$

By extension, an environment of functions $\mathscr{F}$ is aliasing-free when $\mathrm{Env}(\mathscr{F})$ is aliasing-free. □

The notion of aliasing-free environments is not an artifact of our small language, but translates a fundamental property of the C semantics: distinct function parameters or local variables are always bound to distinct memory locations (Section 6.2.2, paragraph 6 in ISO/IEC 9899 [24]).

A local position is any position in a term except inner functions. Local positions are used to distinguish functions defined directly in a term from deeper nested functions, because we need to enforce Invariant 3 (Definition 17) on the former only.

Definition 16 (Local position) *Local positions* are defined inductively as follows:

1. M is in local position in M, $x \coloneqq M$, M ; M, **if** M **then** M **else** M and $f(M, \ldots, M)$.
2. N is in local position in **letrec** $f(x_1 \ldots x_n) = M$ **in** N.

□

We extend the notion of liftable parameter (Definition 11, p. 29) to enforce invariants on stores and environments.

Definition 17 (Extended liftability) The parameter x is *liftable* in $(M, \mathscr{F}, \rho_T, \rho)$ when:

1. x is defined as the parameter of a function g, either in M or in $\mathscr{F}$,
2. in both M and $\mathscr{F}$, inner functions in g, named h_i, are defined and called exclusively:
   (a) in tail position in g, or
   (b) in tail position in some h_j (with possibly $i = j$), or
   (c) in tail position in M,

3. for all f defined in local position in M, $x \in \mathrm{dom}(\rho_T \cdot \rho) \Leftrightarrow \exists i, f = h_i$,
4. moreover, if h_i is called in tail position in M, then $x \in \mathrm{dom}(\rho_T)$,
5. in $\mathscr{F}$, x appears necessarily and exclusively in the environments of the h_i's closures,
6. $\mathscr{F}$ contains only compact closures and $\mathrm{Env}(\mathscr{F}) \cup \{\rho, \rho_T\}$ is aliasing-free.

□

We also extend the definition of lambda-lifting (Definition 9, p. 28) to environments, in order to reflect changes in lambda-lifted parameters captured in closures.

Definition 18 (Lifted form of an environment)

$$\text{If } \mathscr{F}\ f = \left[\lambda x_1 \ldots x_n.b, \rho', \mathscr{F}'\right] \qquad \text{then}$$

$$(\mathscr{F})_*\ f = \begin{cases} \left[\lambda x_1 \ldots x_n x.\,(b)_*, \rho'|_{\mathrm{dom}(\rho')\setminus\{x\}}, (\mathscr{F}')_*\right] & \text{when } f = h_i \text{ for some } i \\ \left[\lambda x_1 \ldots x_n.\,(b)_*, \rho', (\mathscr{F}')_*\right] & \text{otherwise} \end{cases}$$

□

Lifted environments are defined such that a liftable parameter never appears in them. This property will be useful during the proof of correctness.

Lemma 1 *If x is a liftable parameter in $(M, \mathscr{F}, \rho_T, \rho)$, then x does not appear in $(\mathscr{F})_*$.*

Proof Since x is liftable in $(M, \mathscr{F}, \rho_T, \rho)$, it appears exclusively in the environments of h_i. By definition, it is removed when building $(\mathscr{F})_*$. □

These invariants and definitions lead to an enhanced correctness theorem.

Theorem 3 (Correctness of lambda-lifting) *If x is a liftable parameter in $(M, \mathscr{F}, \rho_T, \rho)$, then*

$$M^s \xRightarrow[\mathscr{F}]{\rho_T|\rho} v^{s'} \textit{ implies } (M)_*^s \xRightarrow[(\mathscr{F})_*]{\rho_T|\rho} v^{s'}$$

Since naive and optimised reductions rules are equivalent (Theorem 2, p. 31), the proof of Theorem 1 (p. 29) is a direct corollary of this theorem.

Corollary 1 *If x is a liftable parameter in M, then*

$$\exists t, M^{\varepsilon} \xrightarrow[\varepsilon]{\varepsilon} v^t \textit{ implies } \exists t', (M)_*^{\varepsilon} \xrightarrow[\varepsilon]{\varepsilon} v^{t'}.$$

6.3.2 Overview of the proof

With the enhanced liftability definition, we have strong enough invariants to perform a proof by induction of the correctness theorem. This proof is detailed in Section 6.3.5.

The proof is not by structural induction but by induction on the height of the derivation. This is necessary because, even with the stronger invariants, we cannot apply the induction hypotheses directly to the premises in the case of the (call) rule: we have to change the stores and environments, which means rewriting the whole derivation tree, before using the induction hypotheses.

For that reason, the most important and difficult case of the proof is the (call) rule. We split it into two cases: calling one of the lifted functions ($f = h_i$) and calling another function (either g, where x is defined, or any other function outside of g). Only the former requires rewriting; the latter follows directly from the induction hypotheses.

In the (call) rule with $f = h_i$, issues arise when reducing the body b of the lifted function. During this reduction, indeed, the store contains a new location l' bound by the environment to the lifted variable x, but also contains the location l which contains the original value of x. Our goal is to show that the reduction of b implies the reduction of $(b)_*$, with store and environments fulfilling the constraints of the (call) rule.

To obtain the reduction of the lifted body $(b)_*$, we modify the reduction of b in a series of steps, using several lemmas:

- the location l of the free variable x is moved to the tail environment (Lemma 2);
- the resulting reduction meets the induction hypotheses, which we apply to obtain the reduction of the lifted body $(b)_*$;
- however, this reduction does not meet the constraints of the optimised reduction rules because the location l is not fresh: we rename it to a fresh location l' to hold the lifted variable;
- finally, since we renamed l to l', we need to reintroduce a location l to hold the original value of x (Lemmas 3 and 4).

The rewriting lemmas used in the (call) case are shown in Section 6.3.3.

For every other case, the proof consists in checking thoroughly that the induction hypotheses apply, in particular that x is liftable in the premises. It consists in checking invariants of the extended liftability definition (Definition 17). To keep the main proof as compact as possible, the most difficult cases of liftability, related to aliasing, are proven in some preliminary lemmas (Section 6.3.4).

One last issue arises during the induction when one of the premises does not contain the lifted variable x. In that case, the invariants do not hold, since they assume the presence of x. But it turns out that in this very case, the lifting function is the identity (since there is no variable to lift) and lambda-lifting is trivially correct.

6.3.3 Rewriting lemmas

Calling a lifted function has an impact on the resulting store: new locations are introduced for the lifted parameters and the earlier locations, which are not modified anymore, are hidden. Because of these changes, the induction hypotheses do not apply directly in the case of the (call) rule for a lifted function h_i. We use the following three lemmas to obtain, through several rewriting steps, a reduction of lifted terms meeting the induction hypotheses.

- Lemma 2 shows that moving a variable from the non-tail environment ρ to the tail environment ρ_T does not change the result, but restricts the domain of the store. It is used transform the original free variable x (in the non-tail environment) to its lifted copy (which is a parameter of h_i, hence in the tail environment).
- Lemmas 3 and 4 add into the store and the environment a fresh location, bound to an arbitrary value. It is used to reintroduce the location containing the original value of x, after it has been alpha-converted to l'.

Lemma 2 (Switching to tail environment) *If* $M^s \xrightarrow[\mathscr{F}]{\rho_T|(x,l)\cdot\rho} v^{s'}$ *and* $x \notin \mathrm{dom}(\rho_T)$ *then* $M^s \xrightarrow[\mathscr{F}]{\rho_T\cdot(x,l)|\rho} v^{s'|_{\mathrm{dom}(s')\setminus\{l\}}}$. *Moreover, both derivations have the same height.*

Proof By induction on the structure of the derivation. For the (val), (var), (assign) and (call) cases, we use the fact that $s \setminus \rho_T \cdot (x,l) = s'|_{\mathrm{dom}(s')\setminus\{l\}}$ when $s' = s \setminus \rho_T$. □

Lemma 3 (Spurious location in store) *If* $M^s \xrightarrow[\mathscr{F}]{\rho_T|\rho} v^{s'}$ *and* k *does not appear in either* s, $\mathscr{F}$ *or* $\rho_T \cdot \rho$, *then, for all value* u, $M^{s+\{k\mapsto u\}} \xrightarrow[\mathscr{F}]{\rho_T|\rho} v^{s'+\{k\mapsto u\}}$. *Moreover, both derivations have the same height.*

Proof By induction on the height of the derivation. The key idea is to add (k,u) to every store in the derivation tree. A collision might occur in the (call) rule, if there is some j such that $l_j = k$. In that case, we need to rename l_j to some fresh variable $l'_j \neq k$ (by alpha-conversion) before applying the induction hypotheses. □

Lemma 4 (Spurious variable in environments)

$$\forall l, l', M^s \xrightarrow[\mathscr{F}]{\rho_T\cdot(x,l)|\rho} v^{s'} \quad \textit{iff} \quad M^s \xrightarrow[\mathscr{F}]{\rho_T\cdot(x,l)|(x,l')\cdot\rho} v^{s'}$$

Moreover, both derivations have the same height.

Proof By induction on the structure of the derivation. The proof relies solely on the fact that $\rho_T \cdot (x,l) \cdot \rho = \rho_T \cdot (x,l) \cdot (x,l') \cdot \rho$. □

6.3.4 Aliasing lemmas

We need two lemmas to show that environments remain aliasing-free during the proof by induction in Section 6.3.5. They are purely technical lemmas that consist in proving that the aliasing invariant (Invariant 6, Definition 17) holds in the context of the (call) and (letrec) rules, respectively. We only show the (call) lemma here; the (letrec) lemma is very similar and detailed in the technical report [28].

Lemma 5 (Aliasing in (call) rule) *Assume that, in a (call) rule,*

- $\mathscr{F}\ f = [\lambda x_1 \ldots x_n.b, \rho', \mathscr{F}']$,
- $\mathrm{Env}(\mathscr{F})$ *is aliasing-free, and*
- $\rho'' = (x_1, l_1) \cdot \ldots \cdot (x_n, l_n)$, *with fresh and distinct locations* l_i.

Then $\mathrm{Env}(\mathscr{F}' + \{f \mapsto \mathscr{F}\ f\}) \cup \{\rho', \rho''\}$ *is also aliasing-free.*

Proof Let $\mathscr{E} = \mathrm{Env}(\mathscr{F}' + \{f \mapsto \mathscr{F}\ f\}) \cup \{\rho'\}$. We know that $\mathscr{E} \subset \mathrm{Env}(\mathscr{F})$ so $\mathscr{E}$ is aliasing-free We want to show that adding fresh and distinct locations from ρ'' preserves this lack of freedom. More precisely, we want to show that

$$\forall \rho_1, \rho_2 \in \mathscr{E} \cup \{\rho''\}, \forall x \in \mathrm{dom}(\rho_1), \forall y \in \mathrm{dom}(\rho_2),\ \rho_1\ x = \rho_2\ y \Rightarrow x = y$$

given that

$$\forall \rho_1, \rho_2 \in \mathscr{E}, \forall x \in \mathrm{dom}(\rho_1), \forall y \in \mathrm{dom}(\rho_2),\ \rho_1\ x = \rho_2\ y \Rightarrow x = y.$$

We reason by checking of all cases. If $\rho_1 \in \mathscr{E}$ and $\rho_2 \in \mathscr{E}$, immediate. If $\rho_1 = \rho_2 = \rho''$ then $\rho''\ x = \rho''\ y \Rightarrow x = y$ holds because the locations of ρ'' are distinct. If $\rho_1 = \rho''$ and $\rho_2 \in \mathscr{E}$ then $\rho_1\ x = \rho_2\ y \Rightarrow x = y$ holds because $\rho_1\ x \neq \rho_2\ y$ (by freshness hypothesis). □

6.3.5 Proof of correctness

We finally show Theorem 3.

Theorem 3 (Correctness of lambda-lifting) *If x is a liftable parameter in $(M, \mathscr{F}, \rho_T, \rho)$, then*

$$M^s \xRightarrow[\mathscr{F}]{\rho_T|\rho} v^{s'} \text{ implies } (M)^s_* \xRightarrow[(\mathscr{F})_*]{\rho_T|\rho} v^{s'}$$

Assume that x is a liftable parameter in $(M, \mathscr{F}, \rho_T, \rho)$. The proof is by induction on the height of the reduction of $M^s \xRightarrow[\mathscr{F}]{\rho_T|\rho} v^{s'}$. We only show the case (call). The full proof is available in the technical report [28].

(call) — first case First, we consider the most interesting case where there exists i such that $f = h_i$. The variable x is a liftable parameter in $(h_i(a_1 \dots a_n), \mathscr{F}, \rho_T, \rho)$ hence in $(a_i, \mathscr{F}, \varepsilon, \rho_T \cdot \rho)$ too.

By the induction hypotheses, we get

$$(a_i)_*{}^{s_i} \xRightarrow[(\mathscr{F})_*]{|\rho_T \cdot \rho} v_i^{s_{i+1}}.$$

By definition of lifting, $(h_i(a_1 \dots a_n))_* = h_i((a_1)_*, \dots, (a_n)_*, x)$. But x is not a liftable parameter in $(b, \mathscr{F}', \rho'', \rho')$ since the Invariant 4 might be broken: $x \notin \mathrm{dom}(\rho'')$ (x is not a parameter of h_i) but h_j might appear in tail position in b.

On the other hand, we have $x \in \mathrm{dom}(\rho')$: since, by hypothesis, x is a liftable parameter in $(h_i(a_1 \dots a_n), \mathscr{F}, \rho_T, \rho)$, it appears necessarily in the environments of the closures of the h_i, such as ρ'. This allows us to split ρ' into two parts: $\rho' = (x, l) \cdot \rho'''$. It is then possible to move (x, l) to the tail environment, according to Lemma 2:

$$b^{s_{n+1}+\{l_i \mapsto v_i\}} \xRightarrow[\mathscr{F}'+\{f \mapsto_{\mathscr{F}} f\}]{\rho''(x,l)|\rho'''} v^{s'|_{\mathrm{dom}(s')\setminus\{l\}}}$$

This rewriting ensures that x is a liftable parameter in $(b, \mathscr{F}' + \{f \mapsto_{\mathscr{F}} f\}, \rho'' \cdot (x, l), \rho''')$ (by Lemma 5 for the Invariant 6).

By the induction hypotheses,

$$(b)_*^{s_{n+1}+\{l_i \mapsto v_i\}} \xRightarrow[(\mathscr{F}'+\{f \mapsto_{\mathscr{F}} f\})_*]{\rho''(x,l)|\rho'''} v^{s'|_{\mathrm{dom}(s')\setminus\{l\}}}$$

The l location is not fresh: it must be rewritten into a fresh location, since x is now a parameter of h_i. Let l' be a location appearing in neither $(\mathscr{F}' + \{f \mapsto_{\mathscr{F}} f\})_*$, nor $s_{n+1} + \{l_i \mapsto v_i\}$ or $\rho'' \cdot \rho_T{}'$. Then l' is a fresh location, which is to act as l in the reduction of $(b)_*$.

We will show that, after the reduction, l' is not in the store (just like l before the lambda-lifting). In the meantime, the value associated to l does not change (since l' is modified instead of l).

Lemma 1 implies that x does not appear in the environments of $(\mathscr{F})_*$, so it does not appear in the environments of $(\mathscr{F}' + \{f \mapsto_{\mathscr{F}} f\})_* \subset (\mathscr{F})_*$ either. As a consequence, lack of aliasing implies by Definition 15 that the label l, associated to x, does not appear in $(\mathscr{F}' + \{f \mapsto_{\mathscr{F}} f\})_*$ either, so

$$(\mathscr{F}' + \{f \mapsto_{\mathscr{F}} f\})_*[l'/l] = (\mathscr{F}' + \{f \mapsto_{\mathscr{F}} f\})_*.$$

Moreover, l does not appear in $s'|_{\mathrm{dom}(s')\setminus\{l\}}$. Since l' does not appear in the store or the environments of the reduction, we rename l to l':

$$(b)_*^{s_{n+1}[l'/l]+\{l_i\mapsto v_i\}} \xrightarrow[(\mathscr{F}'+\{f\mapsto \mathscr{F}\ f\})_*]{\rho''(x,l')|\rho'''} v^{s'|_{\mathrm{dom}(s')\setminus\{l\}}}.$$

We want now to reintroduce l. Let $v_x = s_{n+1}\ l$. The location l does not appear in $s_{n+1}[l'/l] + \{l_i \mapsto v_i\}$, $(\mathscr{F}' + \{f \mapsto \mathscr{F}\ f\})_*$, or $\rho''(x,l')\cdot\rho'''$. Thus, by Lemma 3,

$$(b)_*^{s_{n+1}[l'/l]+\{l_i\mapsto v_i\}+\{l\mapsto v_x\}} \xrightarrow[(\mathscr{F}'+\{f\mapsto \mathscr{F}\ f\})_*]{\rho''(x,l')|\rho'''} v^{s'|_{\mathrm{dom}(s')\setminus\{l\}}+\{l\mapsto v_x\}}.$$

Since

$$\begin{aligned}
s_{n+1}[l'/l] + \{l_i \mapsto v_i\} + \{l \mapsto v_x\} &= s_{n+1}[l'/l] + \{l \mapsto v_x\} + \{l_i \mapsto v_i\} && \text{because } \forall i, l \neq l_i \\
&= s_{n+1} + \{l' \mapsto v_x\} + \{l_i \mapsto v_i\} && \text{because } v_x = s_{n+1} l \\
&= s_{n+1} + \{l_i \mapsto v_i\} + \{l' \mapsto v_x\} && \text{because } \forall i, l' \neq l_i
\end{aligned}$$

and $s'|_{\mathrm{dom}(s')\setminus\{l\}} + \{l \mapsto v_x\} = s' + \{l \mapsto v_x\}$, we finish the rewriting by Lemma 4,

$$(b)_*^{s_{n+1}+\{l_i\mapsto v_i\}+\{l'\mapsto v_x\}} \xrightarrow[(\mathscr{F}'+\{f\mapsto \mathscr{F}\ f\})_*]{\rho''(x,l')|(x,l)\cdot\rho'''} v^{\ s'+\{l\mapsto v_x\}}.$$

Hence the result:

$$\frac{\begin{array}{c}(\mathscr{F})_*\ h_i = \left[\lambda x_1 \ldots x_n x.\,(b)_*, \rho', (\mathscr{F}')_*\right] \\ \rho'' = (x_1,l_1)\cdot\ldots\cdot(x_n,l_n)(x,\rho_T\ x) \qquad l' \text{ and } l_i \text{ fresh and distinct} \\ \forall i, (a_i)_*^{\ s_i} \xrightarrow[(\mathscr{F})_*]{|\rho_T\cdot\rho} v_i^{s_{i+1}} \\ (x)_*^{\ s_{n+1}} \xrightarrow[(\mathscr{F})_*]{|\rho_T\cdot\rho} v_x^{s_{n+1}} \qquad (b)_*^{s_{n+1}+\{l_i\mapsto v_i\}+\{l'\mapsto v_x\}} \xrightarrow[(\mathscr{F}'+\{f\mapsto \mathscr{F}\ f\})_*]{\rho''(x,l')|\rho'} v^{\ s'+\{l\mapsto v_x\}}\end{array}}{(h_i(a_1\ldots a_n))_*^{\ s_1} \xrightarrow[(\mathscr{F})_*]{\rho_T|\rho} v^{s'+\{l\mapsto v_x\}\setminus\rho_T}}\ (\textsc{call})$$

Since $l \in \mathrm{dom}(\rho_T)$ (because x is a liftable parameter in $(h_i(a_1\ldots a_n), \mathscr{F}, \rho_T, \rho)$), the extraneous location is reclaimed as expected: $s' + \{l \mapsto v_x\} \setminus \rho_T = s' \setminus \rho_T$.

(call) — second case We now consider the case where f is not one of the h_i. The variable x is a liftable parameter in $(f(a_1\ldots a_n), \mathscr{F}, \rho_T, \rho)$ hence in $(a_i, \mathscr{F}, \varepsilon, \rho_T\cdot\rho)$ too.

By the induction hypotheses, we get

$$(a_i)_*^{\ s_i} \xrightarrow[(\mathscr{F})_*]{|\rho_T\cdot\rho} v_i^{s_{i+1}},$$

and, by Definition 9,

$$(f(a_1\ldots a_n))_* = f((a_1)_*, \ldots, (a_n)_*).$$

If x is not defined in b or $\mathscr{F}$, then $()_*$ is the identity function and can trivially be applied to the reduction of b. Otherwise, x is a liftable parameter in $(b, \mathscr{F}' + \{f \mapsto \mathscr{F}\ f\}, \rho'', \rho')$ — when checking the invariants of Definition 17, we use Lemma 5 for the Invariant 6 and check separately $f = g$ and $f \neq g$ for the Invariants 3 and 4 (see the technical report [28] for more details).

By the induction hypotheses,

$$(b)_*^{s_{n+1}+\{l_i \mapsto v_i\}} \xRightarrow[(\mathscr{F}'+\{f \mapsto \mathscr{F}\, f\})_*]{\rho''|\rho'} v^{s'}$$

hence:

$$(\mathrm{CALL})\ \frac{\begin{array}{c}(\mathscr{F})_*\ f = \left[\lambda x_1 \ldots x_n.\,(b)_*, \rho', (\mathscr{F}')_*\right] \\ \rho'' = (x_1, l_1) \cdot \ldots \cdot (x_n, l_n) \qquad l_i \text{ fresh and distinct} \\ \forall i, (a_i)_*^{\,s_i} \xRightarrow[(\mathscr{F})_*]{|\rho_T \cdot \rho} v_i^{s_{i+1}} \qquad (b)_*^{s_{n+1}+\{l_i \mapsto v_i\}} \xRightarrow[(\mathscr{F}'+\{f \mapsto \mathscr{F}\, f\})_*]{\rho''|\rho'} v^{s'}\end{array}}{(f(a_1 \ldots a_n))_*^{\,s_1} \xRightarrow[(\mathscr{F})_*]{\rho_T|\rho} v^{s'\setminus\rho_T}}$$

Other cases The detailed proof of the other cases is available in the technical report [28]. They are mostly straightforward by induction, except (letrec) which requires an additional aliasing lemma to check the liftability invariants. □

7 Conclusions and further work

In this paper, we have described CPC, a programming language that provides threads which are implemented, in different parts of the program, either as extremely lightweight heap-allocated data structures, or as native operating system threads. The compilation technique used by CPC is somewhat unusual, since it involves a continuation-passing style (CPS) transform for the C programming language. We have shown the correctness of that particular instance of the CPS transform, as well as the correctness of CPC's compilation scheme; while other efficient systems for programming concurrent systems exist, we claim that the existence of a proof of correctness makes CPC unique among them.

CPC is highly adapted to writing high-performance network servers. To prove this fact, we have written Hekate, a large scale BitTorrent seeder in CPC. Hekate has turned out to be a compact, maintainable, fast and reliable piece of software.

We enjoyed writing Hekate very much. Due to the lightweight threads that it provides, and due to the determinacy of scheduling of attached threads, CPC threads have a very different feel from threads in other programming languages; discovering the right idioms and the right abstractions for CPC has been (and remains) one of the most enjoyable parts of our work.

For CPC to become a useful production language it must come equipped with a consistent and powerful standard library that encapsulates useful programming idioms in a generally usable form. The current CPC library has been written on an on-demand basis, mainly to meet the needs of Hekate; the choice of functions that it provides is therefore somewhat random. Filling in the holes of the library should be a fairly straightforward job. For CPC to scale easily to multiple cores, this extended standard library should also offer the ability to run several event loops, scheduled on different cores, and migrate threads between them.

We have no doubt that CPC can be useful for applications other than high-performance network servers. One could for example envision a GUI system where every button is implemented using three CPC threads: one that waits for mouse clicks, one that draws the button, and one that coordinates with the rest of the system. To be useful in practice, such a system should be implemented using a standard widget library; the fact that CPC integrates well with external event loops indicates that this should be possible.

Finally, the ideas used in CPC might be applicable to programming languages other than C. For example, a continuation passing transform might be a way of extending Javascript with threads without the need to deploy a new Javascript runtime to hundreds of millions of web browsers.

Software availability

The full CPC translator, including sources and benchmarking code, is available online at `http://www.pps.jussieu.fr/~kerneis/software/cpc`. The sources of Hekate are available online at `http://www.pps.jussieu.fr/~kerneis/software/hekate`.

References

1. Adya, A., Howell, J., Theimer, M., Bolosky, W.J., Douceur, J.R.: Cooperative task management without manual stack management. In: Proceedings of the 2002 USENIX Annual Technical Conference, pp. 289–302. USENIX Association, Berkeley, CA, USA (2002)
2. Appel, A.W.: Compiling with continuations. Cambridge University Press (1992)
3. Attar, P., Canal, Y.: Réalisation d'un seeder bittorrent en CPC (2009). URL `http://www.pps.jussieu.fr/~jch/software/hekate/hekate-attar-canal.pdf`
4. von Behren, R., Condit, J., Zhou, F., Necula, G.C., Brewer, E.: Capriccio: scalable threads for internet services. SIGOPS Oper. Syst. Rev. **37**(5), 268–281 (2003)
5. Berdine, J., O'Hearn, P., Reddy, U., Thielecke, H.: Linear Continuation-Passing. Higher-Order and Symbolic Computation **15**, 181–208 (2002)
6. Boussinot, F.: FairThreads: mixing cooperative and preemptive threads in C. Concurrency and Computation: Practice and Experience **18**(5), 445–469 (2006)
7. Bruggeman, C., Waddell, O., Dybvig, R.K.: Representing control in the presence of one-shot continuations. In: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation, PLDI '96, pp. 99–107. ACM, New York, NY, USA (1996)
8. Chroboczek, J.: Continuation-passing for C: a space-efficient implementation of concurrency. Tech. rep., PPS, Université Paris 7 (2005). URL `http://hal.archives-ouvertes.fr/hal-00135160/`
9. Claessen, K.: A poor man's concurrency monad. J. Funct. Program. **9**(3), 313–323 (1999)
10. Clinger, W.D.: Proper tail recursion and space efficiency. In: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, PLDI '98, pp. 174–185. ACM, New York, NY, USA (1998)
11. Danvy, O., Schultz, U.: Lambda-lifting in quadratic time. In: Functional and Logic Programming, *Lecture Notes in Computer Science*, vol. 2441, pp. 134–151. Springer-Verlag, Berlin, Germany (2002)
12. Drepper, U., Molnar, I.: The Native POSIX Thread Library for Linux (2005). URL `http://people.redhat.com/drepper/nptl-design.pdf`
13. Duff, T.: Duff's device (1983). URL `http://www.lysator.liu.se/c/duffs-device.html`. Electronic mail to R. Gomes, D. M. Ritchie and R. Pike
14. Dunkels, A., Schmidt, O., Voigt, T., Ali, M.: Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In: Proceedings of the 4th international conference on Embedded networked sensor systems, SenSys '06, pp. 29–42. ACM, New York, NY, USA (2006)
15. Dybvig, R.K., Hieb, R.: Engines from continuations. Comput. Lang. **14**, 109–123 (1989)
16. Engelschall, R.S.: Portable multithreading: the signal stack trick for user-space thread creation. In: Proceedings of the 2000 USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, USA (2000)
17. Fischbach, A., Hannan, J.: Specification and correctness of lambda lifting. J. Funct. Program. **13**, 509–543 (2003)
18. Fischer, J., Majumdar, R., Millstein, T.: Tasks: language support for event-driven programming. In: Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, PEPM '07, pp. 134–143. ACM, New York, NY, USA (2007)
19. Ganz, S.E., Friedman, D.P., Wand, M.: Trampolined style. In: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming, ICFP '99, pp. 18–27. ACM, New York, NY, USA (1999)

20. Haller, P., Odersky, M.: Scala actors: Unifying thread-based and event-based programming. Theoretical Computer Science **410**(2–3), 202–220 (2009)
21. Harris, T., Abadi, M., Isaacs, R., McIlroy, R.: AC: composable asynchronous IO for native languages. In: Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications, OOPSLA '11, pp. 903–920. ACM, New York, NY, USA (2011)
22. Haynes, C.T., Friedman, D.P., Wand, M.: Continuations and coroutines. In: Proceedings of the 1984 ACM Symposium on LISP and functional programming, LFP '84, pp. 293–298. ACM, New York, NY, USA (1984)
23. Hoare, C.A.R.: Monitors: an operating system structuring concept. Commun. ACM **17**(10), 549–557 (1974)
24. International Organization for Standardization: ISO/IEC 9899:1999 "Programming Languages – C" (1999)
25. Johnsson, T.: Lambda lifting: Transforming programs to recursive equations. In: Functional Programming Languages and Computer Architecture, *Lecture Notes in Computer Science*, vol. 201, pp. 190–203. Springer-Verlag, Berlin, Germany (1985)
26. Kerneis, G., Chroboczek, J.: Are events fast? Tech. rep., PPS, Université Paris 7 (2009). URL `http://hal.archives-ouvertes.fr/hal-00434374/`
27. Kerneis, G., Chroboczek, J.: CPC: programming with a massive number of lightweight threads. In: Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-Centric Software, PLACES '11 (2011)
28. Kerneis, G., Chroboczek, J.: Lambda-lifting and CPS conversion in an imperative language. Tech. rep., PPS, Université Paris 7 (2012). URL `http://hal.archives-ouvertes.fr/hal-00669849/`
29. Key, A.: Weave: translated threaded source (with annotations) to fibers with context passing (ca. 1995–2000). As used within RAID-1 code in IBM SSA RAID adapters. Personal communication
30. Krohn, M., Kohler, E., Kaashoek, M.F.: Events can make sense. In: Proceedings of the 2007 USENIX Annual Technical Conference, pp. 1–14. USENIX Association, Berkeley, CA, USA (2007)
31. Landin, P.J.: Correspondence between ALGOL 60 and Church's lambda-notation: part I. Commun. ACM **8**, 89–101 (1965)
32. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The Objective-Caml system (2010). URL `http://caml.inria.fr/`
33. Li, P., Zdancewic, S.: Combining events and threads for scalable network services implementation and evaluation of monadic, application-level concurrency primitives. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation, PLDI '07, pp. 189–199. ACM, New York, NY, USA (2007)
34. Miller, J.S., Rozas, G.J.: Garbage collection is fast, but a stack is faster. AI Memo 1462, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA (1994)
35. Necula, G., McPeak, S., Rahul, S., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Compiler Construction, *Lecture Notes in Computer Science*, vol. 2304, pp. 209–265. Springer-Verlag, Berlin, Germany (2002)
36. Pai, V.S., Druschel, P., Zwaenepoel, W.: Flash: an efficient and portable web server. In: Proceedings of the 1999 USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, USA (1999)
37. Plotkin, G.D.: Call-by-name, call-by-value and the lambda-calculus. Theoretical Computer Science **1**(2), 125–159 (1975)
38. Reppy, J.: Concurrent ML: Design, application and semantics. In: Functional Programming, Concurrency, Simulation and Automated Reasoning, *Lecture Notes in Computer Science*, vol. 693, pp. 165–198. Springer-Verlag, Berlin, Germany (1993)
39. Reynolds, J.C.: The discoveries of continuations. LISP and Symbolic Computation **6**(3), 233–247 (1993)
40. Rompf, T., Maier, I., Odersky, M.: Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. In: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, ICFP '09, pp. 317–328. ACM, New York, NY, USA (2009)
41. Scholz, E.: A concurrency monad based on constructor primitives, or, being first-class is not enough. Tech. Rep. B 95-01, Fachbereich Mathematik und Informatik, Freie Universität Berlin, Berlin, Germany (1995)
42. Shekhtman, G., Abbott, M.: State Threads for internet applications (2009). URL `http://state-threads.sourceforge.net/docs/st.html`
43. Srinivasan, S., Mycroft, A.: Kilim: Isolation-typed actors for java. In: ECOOP 2008 – Object-Oriented Programming, *Lecture Notes in Computer Science*, vol. 5142, pp. 104–128. Springer (2008)
44. Steele Jr., G.L.: Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA (1978). Technical report AI-TR-474
45. Steele Jr., G.L., Sussman, G.J.: Lambda, the ultimate imperative. AI Memo 353, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA, USA (1976)

46. Strachey, C., Wadsworth, C.P.: Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory, Programming Research Group, Oxford, England (1974). Reprinted in Higher-Order and Symbolic Computation 13(1/2):135–152, 2000, with a foreword [51]
47. Tatham, S.: Coroutines in C (2000). URL `http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html`
48. Thielecke, H.: Continuations, functions and jumps. SIGACT News **30**(2), 33–42 (1999)
49. Tismer, C.: Continuations and stackless Python. In: Proceedings of the 8th International Python Conference (2000)
50. Vouillon, J.: Lwt: a cooperative thread library. In: Proceedings of the 2008 ACM SIGPLAN workshop on ML, ML '08, pp. 3–12. ACM, New York, NY, USA (2008)
51. Wadsworth, C.P.: Continuations revisited. Higher-Order and Symbolic Computation **13**(1/2), 131–133 (2000)
52. Wand, M.: Continuation-based multiprocessing. In: Proceedings of the 1980 ACM conference on LISP and functional programming, LFP '80, pp. 19–28. ACM, New York, NY, USA (1980)
53. Welsh, M., Culler, D., Brewer, E.: SEDA: an architecture for well-conditioned, scalable internet services. SIGOPS Oper. Syst. Rev. **35**(5), 230–243 (2001)
54. van Wijngaarden, A.: Recursive definition of syntax and semantics. In: Formal Language Description Languages for Computer Programming, pp. 13–24. North-Holland Publishing Company, Amsterdam, Netherlands (1966)